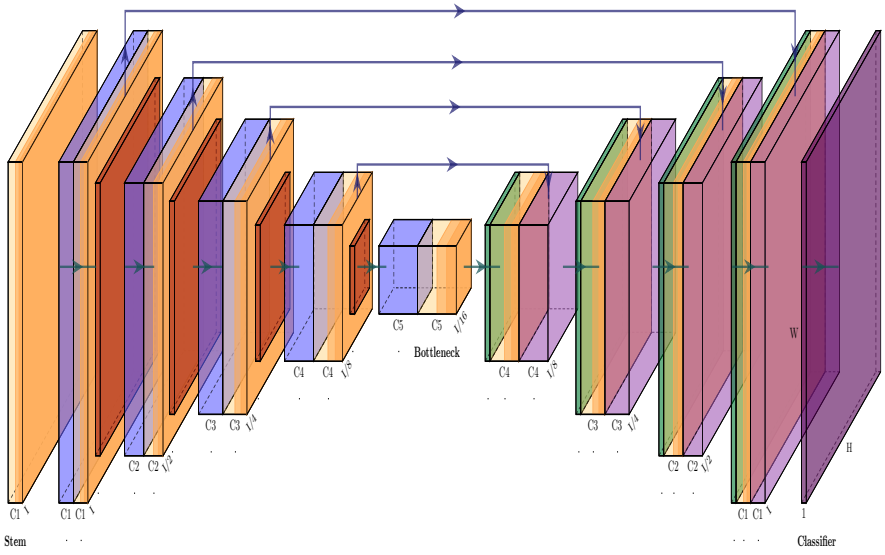


Semantic Segmentation of Medical Images for Fast Diagnosis

ANTÓNIO MARIA FERREIRA DE OLIVEIRA CARVALHO

(Licenciado)

Trabalho de Projeto para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientador:
Doutor Mário Pereira Véstias

Júri:

Presidente:	Doutor Tiago Miguel Braga da Silva Dias
Vogais:	Doutor Mário Pereira Véstias
	Doutor Rui António Policarpo Duarte

Dezembro de 2025

Semantic Segmentation of Medical Images for Fast Diagnosis

ANTÓNIO MARIA FERREIRA DE OLIVEIRA CARVALHO

(Licenciado)

Trabalho de Projeto para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientador:
Doutor Mário Pereira Véstias, IPL/ISEL

Júri:

Presidente:	Doutor Tiago Miguel Braga da Silva Dias, IPL/ISEL
Vogais:	Doutor Mário Pereira Véstias, IPL/ISEL
	Doutor Rui António Policarpo Duarte, IPL/ISEL

Dezembro de 2025

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Mário Véstias, for his continuous guidance, availability, and constructive feedback throughout the development of this work. His insightful suggestions and constant encouragement were invaluable, not only in overcoming technical challenges but also in shaping the direction of this thesis work. I am especially grateful for the time he invested, even during his holidays, always finding a way to provide support and advice. This work would simply not have been possible without his mentorship.

My deepest appreciation also goes to my family — Dad, Mother, and my sisters — for their love, patience, and encouragement. Their support has been a constant source of strength, providing me with the motivation and resilience to carry this work to completion. Every achievement in this dissertation is, in many ways, shared with them.

Finally, I would like to thank my friends — my outlet to the world beyond this work. Your motivation, joy, and perspective carried me through the toughest parts of this journey. The laughter, the spontaneous adventures, and your way of keeping me grounded made the whole experience not just manageable, but genuinely enjoyable.

– António Carvalho

Statement of integrity

I declare that this project work is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Assinado por: **ANTÓNIO MARIA FERREIRA DE OLIVEIRA CARVALHO**
Num. de Identificação: 14895301
Data: 2025.12.19 08:54:36 +0000

Lisbon, December 19, 2025

Semantic Segmentation of Medical Images for Fast Diagnosis

This document was created using the (pdf)LaTeX processor.

Resumo

A segmentação semântica de imagens médicas constitui uma tarefa importante na medicina moderna, permitindo a identificação e localização de estruturas anatómicas e de regiões patológicas ao nível do píxel, o que a torna uma ferramenta fundamental para diagnósticos rápidos e rigorosos. Embora os modelos de aprendizagem profunda tenham alcançado avanços notáveis nesta área, a sua implementação em contextos clínicos reais continua limitada pelas elevadas exigências computacionais das soluções exclusivamente baseadas em software, resultando em processamento ineficiente e com maior latência. Os Field-Programmable Gate Arrays (FPGAs) surgem como uma alternativa promissora ao oferecerem aceleração programável a nível de hardware, possibilitando processamento paralelo, baixa latência e computação energeticamente eficiente, características ideais para aplicações em tempo real, como a deteção de tumores, a delineação de órgãos e a classificação de lesões. Neste trabalho foi desenvolvida e otimizada uma arquitetura de rede neuronal leve, *Mobile-CMUNeXt*, direcionada para implementação em FPGA através de simplificações arquitetónicas, treino quantizado e simplificação de operações, complementada pelo desenvolvimento de núcleos de aceleração modulares para convoluções *depthwise*, *pointwise* e 3D, implementados na plataforma Avnet Ultra96-V2. Os resultados experimentais demonstram que o acelerador proposto atinge desempenho em tempo real com precisão de segmentação clinicamente aceitável, equilibrando de forma eficiente a utilização de recursos e mantendo simultaneamente baixa latência e eficiência energética, além de apresentar uma taxa de processamento competitiva face a implementações em CPU e GPU, validando a eficácia da abordagem de co-design hardware–software. Assim, esta tese comprova que os aceleradores baseados em FPGA constituem uma solução viável para a integração de modelos de aprendizagem profunda em dispositivos médicos de ponta, ao combinar desenho de redes neuronais eficientes com otimização de hardware dedicado, permitindo segmentação semântica de imagens médicas em tempo real, com preservação da privacidade e elevada eficiência energética.

Palavras-chave: Segmentação de imagens médicas; aceleração em FPGA; co-design hardware–software; otimização em aprendizagem profunda; inferência em tempo real.

Abstract

Semantic segmentation of medical images is a critical task in modern healthcare, enabling precise identification and localization of anatomical structures and pathological regions by classifying image regions at the pixel level, thereby supporting rapid and accurate diagnosis. Although deep learning models have achieved remarkable performance in this domain, their deployment in real-world clinical environments is often constrained by the high computational demands of traditional software implementations, resulting in inefficiency and latency. Field-Programmable Gate Arrays (FPGAs) provide a promising alternative by offering programmable hardware-level acceleration tailored for deep learning inference tasks, with advantages such as parallel processing, low latency, and energy-efficient computation, making them particularly suitable for time-sensitive applications including tumor detection, organ delineation, and lesion classification. In this work, a lightweight neural network architecture, *Mobile-CMUNeXt*, was designed and optimized for FPGA deployment through architectural pruning, quantization-aware training, and hardware-friendly modifications, alongside the development of modular accelerator cores for depthwise, pointwise, and 3D convolutions implemented on the Avnet Ultra96-V2 platform. Experimental results demonstrate that the proposed accelerator achieves real-time inference performance with clinically acceptable segmentation accuracy, while balancing resource utilization and maintaining energy efficiency and low latency, and further show competitive throughput compared to CPU and GPU baselines, validating the effectiveness of the hardware–software co-design methodology. Overall, this thesis establishes that FPGA-based accelerators constitute a viable solution for deploying deep learning models in edge medical devices by combining efficient neural network design with custom hardware optimizations, thereby enabling real-time, privacy-preserving, and energy-efficient semantic segmentation of medical images.

Keywords: Medical image segmentation; FPGA acceleration; hardware-software co-design; deep learning optimization; real-time inference.

« Your mind will answer most questions if you learn to relax and wait for the answer »

– William S. Burroughs

Table of Contents

List of Figures

List of Tables

List of Listings

Glossary

AXI4-Lite a lightweight memory-mapped AXI protocol for low-bandwidth control/status registers; used to configure hardware accelerators.

AXI4-Stream a unidirectional streaming protocol in the AXI family for high-throughput data movement without address phases; commonly used for pixel/tensor streams.

bitstream the configuration file programmed into an FPGA to realize a specific design (e.g., `.bit`, `.bin`, or `.xclbin`). It encodes the device configuration for logic, routing, and on-chip resources generated after implementation.

block design a Vivado IP Integrator diagram in which IP blocks are interconnected (e.g., via Advanced eXtensible Interface (AXI)) to form the SoC subsystem, including datapaths, memories, and DMA.

BN folding an export-time optimization that absorbs a Batch Normalization (BN) layer into the preceding convolution by rescaling each output-channel kernel and rebasing the bias: $W'_c = \alpha_c W_c,\ b'_c = (b_c - \mu_c)\alpha_c + \beta_c$ with $\alpha_c = \gamma_c / \sqrt{\sigma_c^2 + \epsilon}$. This removes BN from the runtime graph, reduces memory traffic and arithmetic, and simplifies FPGA datapaths.

BN merge synonym of batch-normalization folding (BN folding): combines BN parameters with the preceding convolution so that inference uses a single Conv→Act sequence with prescaled weights and rebased biases.

co-simulation a verification method that runs C/C++ models alongside their synthesized RTL counterparts to check functional equivalence and interface timing.

dataflow A computational paradigm in which operations are organized as a directed graph of data dependencies, rather than as a sequential list of instructions. In Field Programmable Gate Array (FPGA) design, dataflow refers to structuring hardware so that multiple operations can execute concurrently as soon as their input data is available, enabling pipelined and parallel execution for higher throughput and lower latency..

deep learning a subset of machine learning that uses artificial neural networks with multiple layers to model and learn complex patterns from large amounts of data, commonly used in areas such as image recognition, natural language processing, and autonomous systems.

deep neural network a type of artificial neural network with multiple hidden layers between the input and output layers, enabling the model to learn hierarchical representations of data and capture complex patterns for tasks such as classification, regression, and generation.

depthwise convolution a type of convolution where each input channel is convolved separately with its own filter, preserving the number of input channels in the output and reducing computational cost compared to standard convolution.

energy-efficient A property of a system or architecture that achieves a desired level of performance while minimizing power consumption. In the context of FPGAs and hardware accelerators, energy-efficient designs exploit parallelism, low-precision arithmetic, and on-chip memory to reduce the energy cost per operation, making them suitable for edge and battery-powered devices..

fixed-point value a numerical representation in which real numbers are stored as integers with an implicit scale factor (e.g., $s = 2^{-n}$ for n fractional bits). Signed fixed-point values use two's-complement integers; the represented real is $\hat{x} = I \cdot s$. Fixed-point arithmetic enables low-cost integer operations (adds, shifts) and is widely used on resource-constrained hardware such as FPGAs, especially in quantized neural networks.

floating-point value a numerical representation that encodes a real number as $x = (-1)^s \times m \times b^e$, where s is the sign, m the significand (mantissa), e the exponent, and b the radix (typically 2). Standardized by IEEE 754 (e.g., binary32/single precision and binary64/double precision), floating-point provides a large dynamic range and convenient scaling at the cost of higher area, power, and latency than fixed-point on many accelerators; see also fixed-point value.

forward pass The stage of neural network execution in which input data flows through the layers of the model to produce an output. During this process, operations such as convolution, activation, pooling, and normalization are applied sequentially, but no parameter updates occur. It is typically used in both inference and training (where it precedes the backward pass).

fundus the interior surface of the eye, opposite the lens, including the retina, optic disc, macula, fovea, and posterior pole. Fundus images are commonly used in medical diagnostics to detect conditions such as diabetic retinopathy, glaucoma, and hypertensive retinopathy through non-invasive retinal imaging.

hardware export generation of a hardware platform (e.g., `.xsa`) capturing the configured design, device, and interfaces for use by Vitis applications.

implementation the place-and-route stage that maps a synthesized netlist onto FPGA resources and routing fabric under timing/area constraints.

inference the process of applying a trained machine learning model to new data in order to make predictions or draw conclusions without further training.

laparoscopy A minimally invasive surgical technique in which a surgeon operates through small abdominal incisions using a camera-equipped instrument (laparoscope). It provides real-time visualization of internal organs and allows diagnostic or therapeutic procedures to be performed with reduced patient trauma, faster recovery, and lower risk of complications compared to open surgery.

latency-aware A design principle in which hardware or software is optimized not only for throughput, but also for minimizing end-to-end delay between input and output. In the context of FPGAs, latency-aware architectures explicitly account for pipeline depth, data dependencies, and communication overhead to ensure timely responses, which is critical for real-time and edge applications..

level set A mathematical method for image segmentation that represents object boundaries implicitly as the zero level of a higher-dimensional function, allowing flexible handling of complex shapes, topology changes, and smooth contour evolution.

netlist A *netlist* is a textual or structural representation of an electronic circuit that describes the components (such as logic gates, registers, and arithmetic units) and the interconnections between them. In FPGA and ASIC design, the netlist is typically generated after synthesis and serves as the input for implementation steps such as placement, routing, and power analysis. It reflects the logical hierarchy and connectivity of the design, independent of physical layout..

neural network a computational model inspired by the structure and function of the human brain, consisting of interconnected nodes (neurons) organized in layers, used to recognize patterns and learn from data in tasks such as classification, regression, and decision-making.

on-chip memory FPGA-embedded memories used for buffering and storage (e.g., Block RAM (BRAM) and UltraRAM (URAM)) to reduce off-chip bandwidth.

PDE-based Methods that pose image processing or segmentation as solving a partial differential equation over the image domain (e.g., level sets), often using iterative stencil updates driven by image gradients, curvature, and regularization terms.

pixel the smallest discrete element of an image or display, representing a single point of color or intensity in a digital image; short for "picture element".

quantization metadata Auxiliary information stored alongside quantized parameters or activations, used to correctly interpret their fixed-point representation. Metadata includes bit-width, scale factor, zero-point offset, and sign. In frameworks such as Brevitas, quantization metadata ensures consistent propagation of numerical ranges across layers, enables correct export of fixed-point weights, and preserves alignment between training-time simulation and hardware inference..

quantizer a module that maps floating-point tensors to low-precision integer codes using a scale and (optionally) a zero-point, typically with a fixed bit-width. In QAT frameworks such as Brevitas, a quantizer enforces discretization during training while keeping a floating-point master copy for optimization; it also tracks metadata (bit-width, scale, signedness) so the exported model matches integer-only inference on hardware.

realtime refers to systems or processes that respond to inputs or events within a guaranteed time frame, often instantly or with minimal delay.

semantic segmentation a computer vision technique in which each pixel in an image is classified into a predefined category, enabling detailed understanding of the visual content by identifying object boundaries and regions.

state-of-the-art refers to the most advanced, effective, or sophisticated technique, method, or technology available at a given time in a particular field.

synthesis the tool-driven process that transforms a hardware description into an implementation for a target device. In HLS, C/C++ is compiled to RTL; in logic synthesis, RTL is mapped to FPGA primitives under timing, area, and power constraints, producing a netlist used for place-and-route.

tensor A multi-dimensional array used to represent data in machine learning and deep learning. Scalars (0D), vectors (1D), and matrices (2D) are special cases of tensors. Higher-order tensors (3D, 4D, etc.) are commonly used to store images, batches of images, and feature maps in neural networks.

testbench a verification harness that stimulates a hardware design under test (DUT) with inputs and checks outputs against expected behavior. In FPGA/HLS flows it can be written in C/C++ (for HLS simulation) or HDL (for RTL simulation), and typically includes drivers, monitors, and a reference (golden) model.

toolchain A collection of software tools that are used together in sequence to build, integrate, and deploy a system. In the context of FPGA development, a toolchain typically includes high-level synthesis, hardware design integration, platform packaging, and software application deployment (e.g., Vitis HLS, Vivado, and Vitis). Each stage in the toolchain produces artifacts consumed by the next, forming an end-to-end workflow from high-level code to hardware implementation..

transformer a neural network architecture based on self-attention mechanisms, originally designed for sequence modeling in NLP and later adapted for vision tasks (e.g., ViT). In vision, transformers enable global context modeling across image regions; in language models like ChatGPT, they power advanced reasoning, generation, and understanding through attention-based token interactions.

Vitis Application The final stage of the Xilinx toolchain, where user software is built on top of a Vitis Platform. A Vitis application typically consists of host code running on the processing system and accelerator kernels mapped onto programmable logic. It links hardware and software components together and produces an executable that can be deployed and tested on the target board..

Vitis HLS Xilinx high-level synthesis tool that compiles C/C++/SystemC into RTL, supports C/RTL co-simulation, and packages designs as reusable IP cores.

Vitis Platform a reusable hardware/software target (e.g., `.xsa` with BSP) that defines device, clocks, memory map, and I/O for building and deploying Vitis applications.

Vivado Xilinx tool for RTL synthesis, block design assembly, place-and-route, timing closure, and bitstream generation.

Acronyms

AI Artificial Intelligence

APU Application Processing Unit

ASPP Atrous Spatial Pyramid Pooling

AXI Advanced eXtensible Interface

BatchNorm Batch Normalization

BCE Binary Cross-Entropy

BN Batch Normalization

BRAM Block RAM

BUSI Breast UltraSound Image

CNN Convolutional Neural Network

CRF Conditional Random Field

CT Computed Tomography

CV Computer Vision

DCNN Dilated Convolutional Neural Network

DMA Direct Memory Access

FCN Fully Convolutional Network

FIFO First-In First-Out

FIVES Fundus Image Dataset for Artificial Intelligence based Vessel Segmentation

FN False Negative

FP False Positive

FPGA Field Programmable Gate Array

GPU Graphical Processing Unit

HLS High-Level Synthesis

IoU Intersection over Union

ISIC International Skin Imaging Collaboration

MAC Multiply–Accumulate Operation

MPL Multi-Layer Perceptron

MRI Magnetic Resonance Imaging

NLP Natural Language Processing

PL Programmable Logic

QAT Quantization-Aware Training

RGB Red Green and Blue

RTL Register-Transfer Level

SMC Smart Memory Controller

SoTA State of the Art

TN True Negative

TP True Positive

TPU Tensor Processing Unit

URAM UltraRAM

ViT Vision Transformer

1 Introduction

Today, Artificial Intelligence (AI) surges as the new technological revolution and is being adopted by various sectors [1–3], with healthcare being a critical sector to adopt such technological advancements [4–6]. AI in healthcare encompasses a broad range of applications, from assisting in diagnosing diseases to helping in the predicting patient outcomes, and personalizing treatment plans [7]. In medical diagnosis, AI enhances the accuracy and speeds up the process of image analysis thanks to many deep learning advancements [8], particularly in medical image segmentation. This segmentation process classifies image regions at the pixel level and allows for a wide range of medical applications such as cardiac segmentation from Magnetic Resonance Imaging (MRI) [9], brain tumor segmentation [10], organ segmentation from Computed Tomography (CT) [11–13], retinal vessel segmentation [14] and many more [15]. Figure 1.1 illustrates a typical brain segmentation example with prediction overlaid over the original MRI.

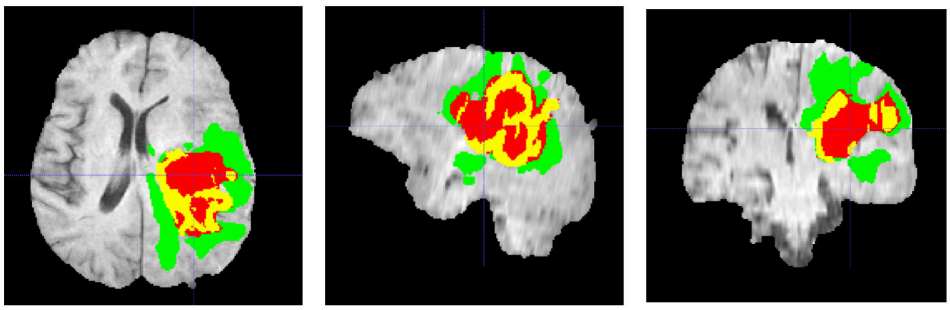


Figure 1.1: Typical MRI brain segmentation with prediction overlaid over original, adapted from [10]. Red, Green And Yellow regions represent different predicted classes.

The incorporation of semantic segmentation into the medical image diagnoses process is a game-changer for healthcare enhancing both accuracy and efficiency. These models improve diagnostic accuracy as they are trained on vast datasets, allowing them to recognize patterns and anomalies that might be imperceptible to the human eye. For example, in dermatology, AI systems trained on images of skin lesions have shown the ability to detect skin cancers, such as melanoma, with a level of precision comparable to that of experienced dermatologists [16]. Also, these systems help speed up medical images analysis and test results, as tasks that would take a healthcare professional considerable time, can be performed in a fraction of the time. This rapid analysis is particularly beneficial in urgent care situations, where quick decision-making is critical.

Ultimately, the role of AI in improving diagnostic accuracy and efficiency is transforming health-

care, delivering faster and more precise diagnoses. However, this transformation hinges on the immense processing power required to analyze large medical datasets along with the complex algorithms used for these semantic segmentation tasks — this is where hardware acceleration steps in.

Semantic segmentation networks continue to achieve impressive results, but they are also becoming larger and less computationally efficient, posing challenges for realtime applications. FPGAs play a crucial role, allowing for the design of these networks in programmable hardware, these devices enable hardware level parallel processing which makes them highly efficient for accelerating deep learning inference tasks. Unlike pre-designed Graphical Processing Units (GPUs) and Tensor Processing Units (TPUs), FPGAs can be programmed to perform specific algorithms with minimal latency and power consumption.

1.1 Motivation

Semantic segmentation of medical images has transformed diagnostic workflows, but in many clinical contexts, real-time performance is essential for safe and effective decision-making. For example, during intraoperative brain tumor resection, surgeons rely on live ultrasound or MRI to adapt to changes in anatomy caused by tissue deformation ("brain shift"), which renders static pre-operation images less reliable as the surgery progresses [17, 18]. Real-time segmentation in these workflows can continuously update tumor boundaries and help avoid damage to functional brain regions [19, 20]. Similar needs arise in laparoscopy and robotic surgery, where segmentation of critical anatomical structures — such as nerves, vessels, and ducts — must be available with minimal delay to prevent injury while tools navigate constrained surgical fields. In other acute care scenarios, such as emergency radiology, faster segmentation of lesions and hemorrhages in CT images improve triage and intervention times, directly impacting patient outcomes. Such cases demand segmentation results in milliseconds to seconds rather than minutes or hours.

Despite the success of deep learning models in producing accurate segmentation maps offline, their deployment in real-time clinical systems confronts significant constraints. Conventional processors such as CPUs and GPUs are designed for throughput rather than low latency, and often require energy, size, and cooling budgets unsuited for embedded or portable medical devices. FPGAs offer a promising alternative: their reconfigurable logic enables highly parallel pipelines customized for specific inference tasks [21].

FPGA-accelerated approaches have demonstrated real-time performance advantages in medical contexts. For instance, FPGA implementations of neural networks for diagnostic analysis have shown dramatic reductions in latency compared to CPU and GPU baselines, enabling direct sensor interfacing and low-latency feedback during procedures [22]. Additionally, dedicated FPGA accelerators for brain tumor segmentation achieve speedups of over an order of magnitude while significantly improving energy efficiency [23, 24]. These results suggest that FPGA platforms can fulfill the stringent timing and resource constraints of real-time medical imaging applications.

Taken together, the clinical urgency of real-time image interpretation in operative and emer-

gency settings, the architectural strengths of FPGAs, and the demonstrated hardware acceleration potential underscore the need for optimized segmentation models tailored for FPGA deployment. This thesis builds on these motivations by exploring low-latency, energy-efficient network designs suitable for real-time medical image segmentation on reconfigurable hardware.

1.2 Objectives

The primary objective of this work is to develop, optimize, and deploy a deep neural network on an FPGA for realtime medical image segmentation inference. To achieve this goal, a comprehensive state-of-the-art analysis was conducted to identify suitable segmentation models and datasets. Based on this analysis, a lightweight network architecture was designed to reduce model complexity while preserving segmentation accuracy under strict hardware constraints. Next, a modular and testable hardware architecture was developed to support efficient deployment. Finally, the FPGA implementation was validated against the original software model to ensure functional equivalence and performance consistency. The proposed *Mobile-CMUNeXt* architecture leverages pruning and quantization-aware training to minimize computational and memory requirements, enabling low-latency inference on resource-constrained devices.

1.3 Research and Publications

This section presents the research outputs of this master's thesis, including peer-reviewed publications and publicly available source code, thereby supporting transparency, reproducibility, and open scientific research.

The complete source code developed and used throughout this work — including implementations for training, evaluation, and experimentation in medical image semantic segmentation — is publicly available at `https://github.com/ACRae/Mobile-CMUNeXt`.

Published Papers:

1. A. Carvalho and M. Véstias, "Fast Semantic Segmentation of Medical Images," in *Proceedings of the 9th International Young Engineers Forum on Electrical and Computer Engineering (YEF-ECE)*, Caparica/Lisbon, Portugal, 2025, pp. 56–61, doi: 10.1109/YEF-ECE66503.2025.11117287.
2. A. O. Carvalho, M. Véstias, "Lightweight Semantic Segmentation of Medical Images for Embedded Healthcare Systems," submitted to *IEEE Transactions on Biomedical Circuits and Systems (TBioCAS)*, Manuscript ID: TBioCAS-2025-Nov-0413-Reg, 2025.

1.4 Thesis Structure

The remainder of this thesis is organized into six chapters, each addressing a specific aspect of the research, ranging from theoretical background to FPGA deployment.

- **Chapter 2: Background and Related Work** introduces the foundations of convolutional

neural networks and surveys the main families of semantic segmentation architectures. Improvements to the U-Net design and related advances are also discussed.

- **Chapter 3: Model Exploration** describes the experimental environment adopted in this work. The datasets, evaluation metrics, and initial benchmarks across candidate networks are presented. A systematic criterion for model selection is proposed, leading to the identification of a final candidate.
- **Chapter 4: Mobile-CMUNeXt** details the optimizations applied to the selected architecture to make it more suitable for FPGA implementation. The final optimized Mobile-CMUNeXt design and its experimental results are reported.
- **Chapter 5: Mobile-CMUNeXt Quantization** focuses on adapting the chosen network to fixed-point arithmetic. It covers the setup for training, batch normalization folding, fine-tuning, and weight export. The chapter also reports the outcomes of quantization experiments and their effect on accuracy.
- **Chapter 6: Hardware Acceleration** outlines the end-to-end FPGA deployment of the quantized Mobile-CMUNeXt model. The design of the accelerator cores is described, highlighting their resource-efficient implementation and integration into the system.
- **Chapter 7: Conclusion** summarizes the main contributions of this thesis, reflects on the achieved results, and discusses potential directions for future research.

2 Background and Related Work

In the field of Computer Vision (CV), semantic segmentation enables machines to interpret the structure of visual data. This chapter describes the crucial components that make up semantic segmentation neural networks. The understanding of the fundamental semantic segmentation layers is essential for the developed project. Secondly a brief state-of-the-art is conducted, introducing the most effective segmentation network architectures. Moreover, some improvements to these networks are described. Finally, typical neural networks for medical imaging are described and a brief state-of-the-art of FPGA implementations for semantic segmentation is presented.

2.1 Semantic Segmentation

Semantic segmentation is a technique utilized for classifying image regions at the pixel level allowing for the distinction of elements in the background from elements in the foreground. This feature extraction is very important in the medical sector as it provides aided diagnoses and allows for automated medical imaging analysis [4, 5]. Semantic segmentation is a challenging task as it requires both high-level understanding of the image content and precise localization of objects. The main goal of semantic segmentation is to assign a class label to each pixel in an image, effectively partitioning the image into meaningful segments corresponding to different objects or regions. This is typically achieved using deep learning techniques, particularly convolutional neural networks (Convolutional Neural Networks (CNNs)), which are well-suited for processing visual data. Semantic segmentation has a wide range of applications, including autonomous driving, medical imaging, and scene understanding.

2.2 Convolutional Neural Networks

CNNs are a class of deep neural network that have proven highly effective for a wide range of vision tasks, including image classification, detection, and semantic segmentation [25–27]. They learn spatial hierarchies of features through stacked convolutional layers, activation functions, pooling and normalization layers.

2.2.1 Convolutional layers

Convolution layers are the core building blocks of CNNs. They apply a set of learnable filters (or kernels) to the input data (tensor), producing feature maps that capture spatial hierarchies

and patterns. Each filter is designed to detect specific features, such as edges or textures, by sliding over the input and performing element-wise multiplications followed by summations. Formally, given an input map f and a convolutional kernel k, the output feature map y is defined as:

$$y = \sum_i f_i * k_i + b$$

where f_i denotes the input values, k_i the kernel weights, and b is a learnable bias parameter. The convolution operation $(*)$ slides the kernel across the input map, computing weighted sums at each position. The bias term shifts the output, allowing greater flexibility in the learned features.

This operation naturally extends to multichannel inputs, such as RGB images with three channels. In such cases, the kernel has a corresponding set of weights per channel, and the outputs are summed to form the final feature map. Figure 2.1 illustrates how a convolution is applied to a RGB image.

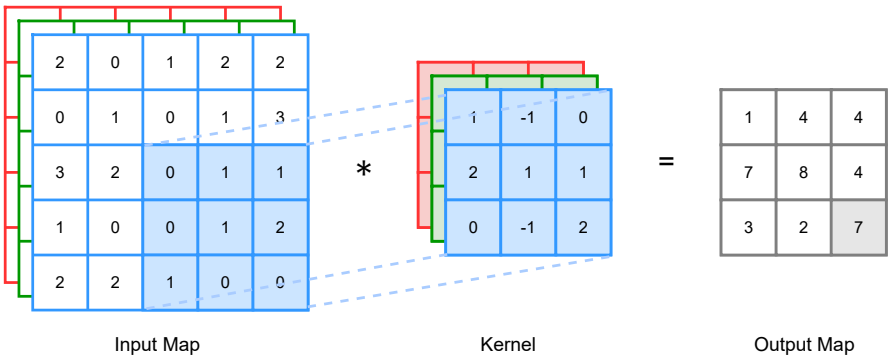


Figure 2.1: Typical convolution operation. Input map has $5 \times 5 \times 3$ dimensions, the kernel has $3 \times 3 \times 3$ dimensions, and the output map has $3 \times 3 \times 1$ dimensions

The output feature map dimensions for convolutions can be computed using the following formula:

$$H_{out} = \left\lfloor \frac{H_{in} + 2P - K}{S} \right\rfloor + 1, \quad W_{out} = \left\lfloor \frac{W_{in} + 2P - K}{S} \right\rfloor + 1$$

where P is padding, K is kernel size, S is stride, and this formula applies identically for each group.

Bias is a trainable value that is added to the output of the convolution operation. There is a bias for each output channel, this bias is then added to the values of each map. It allows for the model to shift the output of the convolution with the intent of fitting the data better especially for cases when the output value is zero.

Padding is added to the input in order to control the spatial dimensions of the output after the convolutional operation. Without padding, each convolution reduces the width and height of the output. In Figure 2.2 a 2×2 filter reduces the size of a 3 input padded to 5×5 to an output map of 4×4.

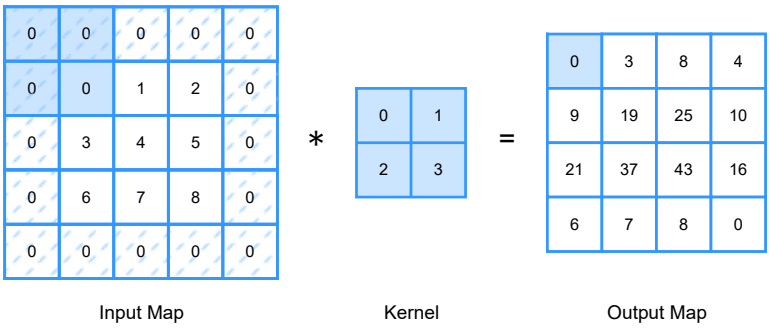


Figure 2.2: Convolution with input padding

Stride defines the step size at which the convolutional kernel moves across the input map. A stride of 1 means the kernel moves one pixel at a time, while a stride of 2 means it moves two pixels at a time. Increasing the stride reduces the spatial dimensions of the output feature map, effectively downsampling the input.

Groups directly affect the structure and efficiency of convolution operations by controlling how input channels are partitioned and processed. A typical convolution defines groups as $groups = 1$. In cases where $groups > 1$ the convolution operation is decomposed into parallel, independent sub-operations, each handling a subset of input channels. More specifically in cases where $groups = C_{in}$, where C_{in} represents input channels, the operation is commonly called **depthwise convolution**, Figure 2.3 illustrates a depthwise convolution. Increasing the number of groups reduces complexity as each filter is applied to fewer input channels, decreasing both the number of multiplications required and the total learnable parameters.

2.2.2 Dilated Convolutions

Dilated Convolutions or atrous convolutions introduce gaps between the kernel elements, allowing the convolution to cover a larger area of the input without increasing the number of parameters or the amount of computation. This is particularly useful for capturing multiscale context in images. The dilation rate determines the spacing between kernel elements, with a dilation rate of 1 corresponding to standard convolution. For example, a 3×3 kernel with a dilation rate of 2 effectively covers a 5×5 area of the input.

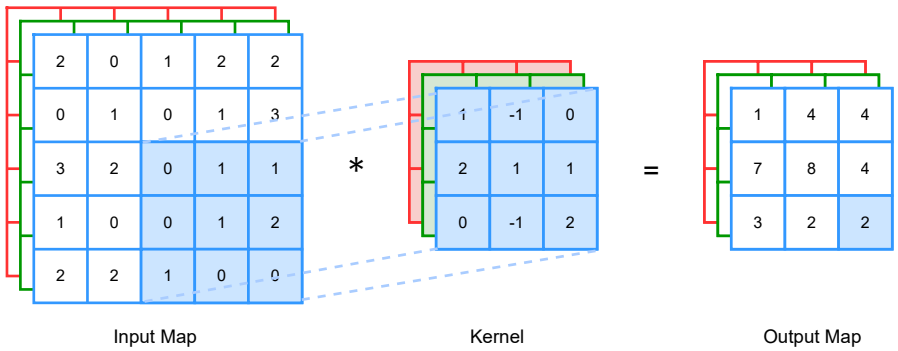


Figure 2.3: Depthwise convolution, each input channel is convolved independently

2.2.3 Upsampling

Upsampling is a technique used to increase the spatial resolution of feature maps, which is particularly important in tasks like semantic segmentation where precise localization is required. Common upsampling methods include:

- **Nearest Neighbor Interpolation**: This method assigns the value of the nearest pixel to the new pixel location. It is simple and fast but can result in blocky artifacts.
- **Bilinear Interpolation**: This method computes the value of a new pixel based on a weighted average of the four nearest pixels. It provides smoother results than nearest neighbor interpolation.
- **Bicubic Interpolation**: This method uses the values of the 16 nearest pixels to compute a new pixel value, resulting in even smoother images than bilinear interpolation.
- **Transposed Convolution (Deconvolution)**: This learnable upsampling method uses convolutional kernels to increase the spatial dimensions of feature maps. It allows the network to learn optimal upsampling filters during training, often leading to better performance in segmentation tasks.

2.2.4 Activation Functions

Activation functions introduce non-linearity into neural networks, enabling them to learn complex patterns. Common activation functions include:

- **ReLU (Rectified Linear Unit)**: Defined as $f(x) = \max(0, x)$, it is widely used due to its simplicity and effectiveness in mitigating the vanishing gradient problem.
- **Leaky ReLU**: A variant of ReLU that allows a small, non-zero gradient when the input is negative, defined as $f(x) = \max(\alpha x, x)$ where α is a small constant.
- **PReLU (Parametric ReLU)**: An extension of Leaky ReLU where the slope of the negative part is learned during training, defined as $f(x) = \max(\alpha x, x)$ with α being a learnable parameter.
- **Sigmoid**: Maps inputs to the range (0, 1), defined as $f(x) = \frac{1}{1+e^{-x}}$. It is often used in binary classification tasks.
- **Tanh**: Maps inputs to the range (-1, 1), defined as $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$. It is zero-centered, which can help with convergence.
- **Softmax**: Converts a vector of values into a probability distribution, defined as $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$. It is commonly used in multi-class classification tasks.

2.2.5 Batch Normalization

Batch Normalization (BatchNorm) is a technique used to stabilize and accelerate the training of deep neural networks. It normalizes the inputs of each layer to have a mean of zero and a

standard deviation of one, which helps mitigate issues like internal covariate shift — the change in the distribution of network activations due to the updating of parameters during training. The normalization is performed over a mini-batch of data, hence the name. The batch normalization process involves the following steps:

- Compute the mean and variance of the mini-batch.
- Normalize the inputs using the computed mean and variance.
- Scale and shift the normalized inputs using learnable parameters (gamma and beta).
- During inference, the mean and variance are replaced with running averages computed during training.

The batch normalization operation can be mathematically expressed as:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y = \gamma \hat{x} + \beta \tag{2.1}$$

where x is the input, μ and σ^2 are the mean and variance of the mini-batch, ϵ is a small constant to prevent division by zero, and γ and β are learnable parameters that scale and shift the normalized output. Batch normalization has several benefits, including improved gradient flow, reduced sensitivity to weight initialization, and the ability to use higher learning rates. It also acts as a form of regularization, potentially reducing the need for other regularization techniques like dropout.

2.2.6 Pooling

Pooling layers reduce the spatial dimensions of feature maps, helping to decrease computational load and control overfitting. Common pooling methods include:

- **Max Pooling**: This method selects the maximum value from a defined window (e.g., 2×2) and stride, effectively downsampling the feature map while retaining the most prominent features.
- **Average Pooling**: This method computes the average value within a defined window and stride, providing a smoother downsampling compared to max pooling.
- **Global Average Pooling**: This method computes the average value across the entire spatial dimensions of the feature map, reducing it to a single value per channel. It is often used before fully connected layers in classification tasks.

2.3 Complexity of deep neural networks

CNN models can be computationally intensive, especially for real-time applications or deployment on resource-constrained devices. Two key metrics used to evaluate the efficiency of these models are the number of parameters and the number of Multiply–Accumulate Operations (MACs).

MAC is a common metric used to quantify the computational complexity of convolutional operations. It stands for Multiply–Accumulate Operation, which involves multiplying two numbers and adding the result to an accumulator. In the context of neural networks, MACs are used to measure the number of such operations required to process an input through a layer, particularly in convolutional layers. The total number of MACs for a convolutional layer can be calculated using Equation 2.2:

$$\text{MACs} = H_{out} \times W_{out} \times C_{out} \times (C_{in} \times K^2) \tag{2.2}$$

H_{out} and W_{out} are the height and width of the output feature map, C_{out} is the number of output channels, C_{in} is the number of input channels, and K is the kernel size. This metric is crucial for evaluating the efficiency of neural network architectures, especially in resource-constrained environments.

The number of parameters in a neural network refers to the total count of learnable weights and biases within the model. Each layer in the network contributes to this count based on its architecture. For instance, in a convolutional layer, the number of parameters can be calculated as Equation 2.3:

$$\text{Parameters} = (C_{in} \times K^2 + 1) \times C_{out} \tag{2.3}$$

where C_{in} is the number of input channels, K is the kernel size, and C_{out} is the number of output channels. The "+1" accounts for the bias term associated with each output channel. The total number of parameters in a neural network is the sum of parameters across all layers. This metric is important as it directly impacts the model's capacity to learn from data, its memory footprint, and its computational requirements during both training and inference.

2.4 Common Neural Networks for Segmentation

This section reviews the most influential and common neural network architectures that have shaped the field of segmentation research. It explores how these models address core challenges, such as preserving spatial detail and capturing multi-scale context. The architectures discussed mark milestones in the evolution of segmentation techniques each offering distinct contributions and innovations.

Fully Convolutional Network (FCN) [27] was the first deep learning-based architecture designed explicitly for semantic segmentation. It replaces fully connected layers with convolutional layers, allowing dense predictions from inputs of arbitrary size. FCN introduces skip connections that fuse coarse semantic information with finer spatial details using bilinear upsampling and deconvolution layers, improving the resolution and accuracy of segmentation outputs. Figure 2.4 illustrates FCN architecture.

SegNet [28] is an encoder-decoder architecture designed for efficient semantic segmentation, particularly in scene understanding tasks. The encoder mirrors the VGG16 [29] convolutional layers, while the decoder uses the pooling indices from the encoder to upsample feature maps, avoiding the need to learn upsampling weights. This approach reduces computational complexity and memory usage, making SegNet more efficient than FCN. Figure 2.5 illustrates SegNet architecture.

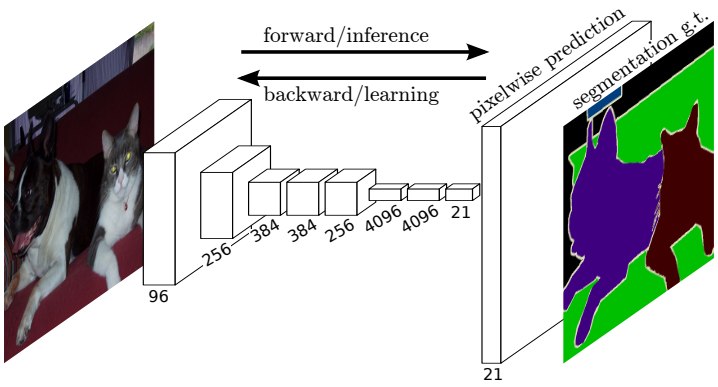


Figure 2.4: FCN architecture. Copyright 2015, Long et al. [27]

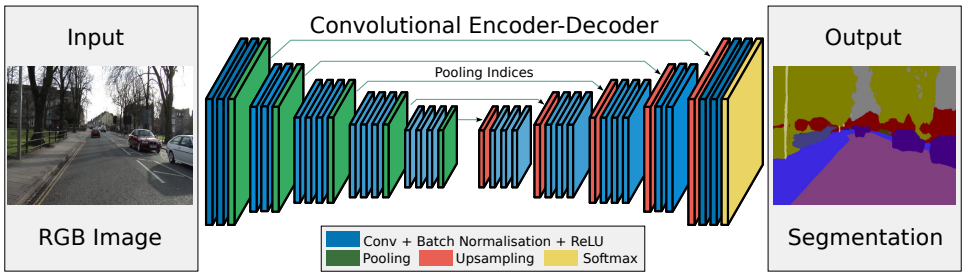


Figure 2.5: SegNet architecture. Copyright 2015, Badrinarayanan et al. [28]

U-Net [30] was developed for biomedical image segmentation and is notable for its symmetric U-shaped architecture. It consists of a contracting path that captures context and an expansive path that enables precise localization. Skip connections concatenate high-resolution features from the encoder with upsampled outputs in the decoder, enhancing segmentation accuracy even with limited training data. Figure 2.6 illustrates U-Net architecture.

The left side of the figure represents the contracting path, which consists of repeated applications of two 3×3 convolutions, each followed by a ReLU activation and a 2×2 max pooling operation with stride 2 for downsampling. This path captures the context of the input image by progressively reducing its spatial dimensions while increasing the number of feature channels. The right side of the figure represents the expansive path, which consists of upsampling the feature map followed by a 2×2 convolution (up-convolution) that halves the number of feature channels. This is followed by concatenation with the corresponding feature map from the contracting path via skip connections, and two 3×3 convolutions each followed by a ReLU activation. This path enables precise localization by combining high-resolution features from the contracting path with the upsampled features.

V-Net [31] extends the U-Net architecture for volumetric (3D) medical image segmentation. Designed to operate on full 3D volumes such as CT or MRI scans, it uses an encoder-decoder structure with 3D convolutions. Key innovations include the use of residual connections (skip connections that directly add the input of a layer (or block of layers) to its output) for improved training stability, strided convolutions instead of max-pooling for downsampling, and Parametric ReLU activations. V-Net enables richer spatial context capture and accurate segmentation of complex anatomical structures. Figure 2.7 illustrates V-Net architecture.

DeepLab [32] introduced several major innovations for semantic segmentation. First, it employs

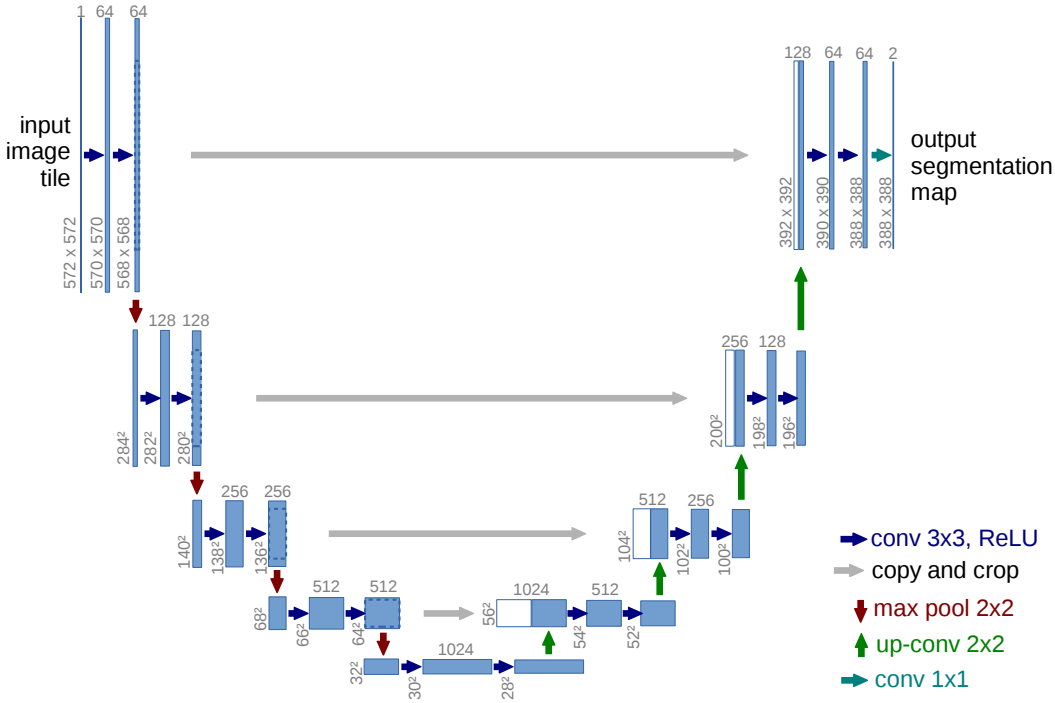


Figure 2.6: U-Net architecture. Copyright 2015, Ronneberger et al. [30]

dilated convolutions to expand the receptive field without increasing computation. Second, it introduces Atrous Spatial Pyramid Pooling (ASPP) to capture multi-scale context by applying atrous convolutions at multiple rates. Finally, DeepLab enhances boundary localization by integrating a fully connected Conditional Random Field (CRF) — a probabilistic model that refines per-pixel labels by enforcing spatial consistency and aligning boundaries to image edges — with CNN outputs, improving segmentation accuracy around object edges. Figure 2.8 provides an overview of DeepLab network.

The input image is processed by a Dilated Convolutional Neural Network (DCNN) that uses atrous convolutions arranged in an ASPP module to enlarge the receptive field while keeping a dense feature map. The network produces a per-class score map, which is then upsampled via bilinear interpolation to recover the original resolution. Because interpolation blurs boundaries, a fully connected CRF is applied next, refining the upsampled scores by encouraging nearby, visually similar pixels to share labels.

2.5 Improvements to U-Net

This section examines architectural enhancements and optimization techniques that aim to improve upon the original U-Net. These improvements focus on increasing segmentation accuracy, reducing computational complexity, and enhancing the model's ability to capture multi-scale context. The discussed advancements include the integration of transformer modules, the adoption of Multi-Layer Perceptron (MPL) blocks, and the use of depthwise separable convolutions.

The transformers were originally developed for Natural Language Processing (NLP) but have since been successfully ported for various CV tasks. These attention-based architectures

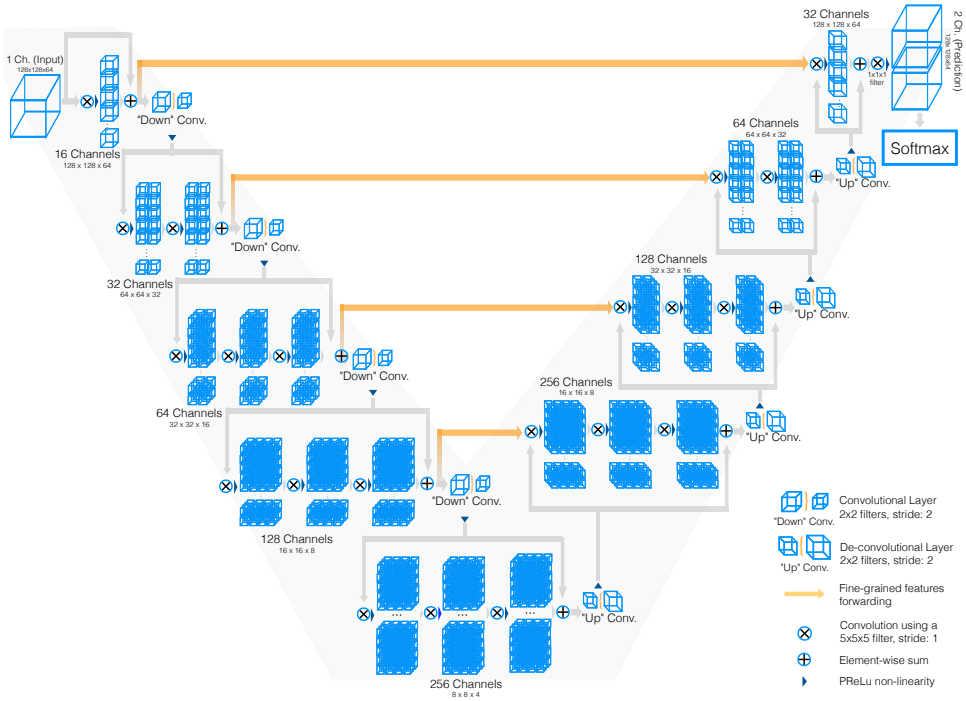


Figure 2.7: V-Net architecture. Copyright 2016, Milletari et al. [31]

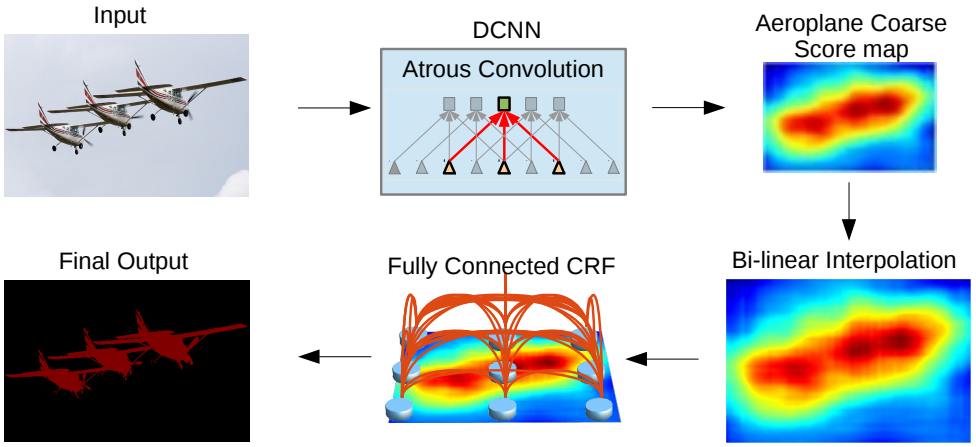


Figure 2.8: DeepLab architecture overview. Copyright 2017, Chen et al. [32]

have become very popular in the computer vision field by enabling global context extraction. The Vision Transformer (ViT) [33] introduced an alternative based on self-attention layers for sequence-to-sequence prediction achieving state-of-the-art performance.

ViT based architectures mostly focus on improving network performance and precision but often overlook aspects like computational complexity, inference time and number of parameters. The MPL blocks have emerged as an alternative to the self-attention mechanisms proposed in transformer based architectures, by combining efficient feature extraction with reduced computational effort.

The MLP-Mixer [34–37] introduced a technically simpler architecture that replaces convolutions and self-attention with token-mixing and channel-mixing layers based exclusively on MPLs, demonstrating that competitive performance could be obtained without relying on self-attention mechanisms or convolutions.

Depthwise separable convolutions are designed to enhance computational efficiency and reduce the number of parameters in deep learning models. This design separates a convolution into two operations: a depthwise convolution, which applies a single convolutional filter per input channel, followed by a pointwise convolution, which uses a 1×1 convolution to combine the outputs of the depthwise convolution across channels. This design was popularized by MobileNetV2 [38] and proved to be very effective for lightweight deep neural networks.

Computational cost. The depthwise–separable convolution significantly reduces the number of MACs compared to a standard convolution. This reduction is particularly beneficial for real-time applications and deployment on resource-constrained devices. The computational savings can be quantified by comparing the MAC counts of standard and depthwise–separable convolutions and the number of learnable parameters. Let the input have C_{in} channels, the output C_{out} channels, kernel size $k \times k$, and spatial size $H \times W$. The MAC cost of a standard convolution is given by Equation 2.4:

$$\text{MACs}_{\text{std}} \;=\; HW\, C_{\text{in}} C_{\text{out}} k^2. \tag{2.4}$$

For a depthwise–separable convolution (depthwise $k \times k$ followed by pointwise 1×1), the cost becomes Equation 2.5:

$$\text{MACs}_{\text{dw+pw}} \;=\; HW\big(C_{\text{in}} k^2 + C_{\text{in}} C_{\text{out}}\big). \tag{2.5}$$

Dividing Equation 2.5 by Equation 2.4 yields the reduction factor in Equation 2.6:

$$\frac{\text{MACs}_{\text{dw+pw}}}{\text{MACs}_{\text{std}}} \;=\; \frac{1}{C_{\text{out}}} \;+\; \frac{1}{k^2}. \tag{2.6}$$

As Equation 2.6 highlights, savings grow with larger C_{out} and kernel size k; for common segmentation settings, the depthwise–separable form requires only a small fraction of the MACs of a standard convolution.

The number of learnable parameters is another important metric for evaluating model efficiency. Depthwise separable convolutions reduce the parameter count significantly compared to standard convolutions, which is particularly advantageous for deployment on devices with limited memory capacity. The parameter count for a standard convolution is given by Equation 2.7:

$$P_{\text{std}} \;=\; C_{\text{in}} C_{\text{out}} k^2. \tag{2.7}$$

The depthwise–separable version uses the parameterization in Equation 2.8:

$$P_{\text{dw+pw}} \;=\; C_{\text{in}} k^2 \;+\; C_{\text{in}} C_{\text{out}}. \tag{2.8}$$

Comparing Equation 2.8 with Equation 2.7 makes clear that removing cross–channel spatial kernels and deferring channel mixing to a 1×1 projection yields substantially fewer weights, typically with minimal loss in representational power for dense prediction tasks.

2.6 Neural Networks For Medical Images

In this section several neural network architectures for medical image segmentation are presented, these are divided into different categories. The first category, *U-Net-based models*, includes architectures that build directly upon the original U-Net, incorporating small modifications to enhance performance. The second category, *Transformer-based models*, highlights recent advancements that integrate self-attention mechanisms and transformer modules to capture global features in medical images. Finally, the *Lightweight variants'* category focuses on models designed for efficient inference, reducing parameter counts and MACs while providing new strategies to preserve feature and segmentation accuracy.

UNet++ [39] refines the skip connections with nested dense convolutional blocks, allowing better feature fusion between encoder and decoder stages. UNetV2 [11] further improves segmentation by incorporating normalization, deep supervision, and better regularization techniques. Attention U-Net [40] integrates attention gates into the skip connections, enhancing the network's focus on relevant anatomical regions during decoding.

In the context of medical image semantic segmentation, several transformer-enhanced variants of the original U-Net architecture have appeared, integrating self-attention mechanisms to enhance performance and precision. TransUnet [41] combines CNNs with transformer, Swin-UNet [42] leverages hierarchical Swin Transformer blocks, SwinMM [13] integrates masked multi-view with Swin Transformers for 3D medical image segmentation, and many more [43].

Several networks have been proposed in order to achieve compact and efficient medical image segmentation capable of handling complex tasks while reducing the number of parameters and computational cost.

As already seen, UNeXt provides efficient medical image segmentation based on MLP blocks and skip connections.

For multimodal biomedical image segmentation, CFPNet-M [14] is a lightweight encoder-decoder-based network specifically developed for real-time segmentation using feature pyramid channels.

Dinh et al. [44] demonstrated that U-Lite, a CNN-based model with only one million parameters, is sufficient for medical image segmentation. Al-Fahsi et al. [45] proposed GIVTED-Net, a network combining GhostNet [46], Involution [47], and ViT for lightweight medical image segmentation. She et al. [48] introduced LUCF-Net, a lightweight U-shaped cascade fusion network for medical image segmentation, optimizing multiscale feature extraction.

Lastly, Tang et al. [49] introduced CMUNeXt, an efficient medical image segmentation network leveraging depthwise separable convolutions from MobileNetV2 to enhance performance. It integrates concepts from ConvUNext [50] and ConvMixer [51] to combine the structural advantages of large kernel convolutions with depthwise separability.

All these lightweight networks show the same trend, that it's possible to cut down parameters and computation without losing too much segmentation accuracy. With many designs reaching strong results, providing clear evidence that lightweight approaches are a valid direction for

medical image segmentation, helping to further support and solidify the work done in this thesis.

2.7 FPGA Implementations for Semantic Segmentation

FPGA-based solutions for semantic segmentation aim to deliver low-latency, energy-efficient inference by tailoring compute, memory access, and dataflow to the structure of the model. Below we review representative works that, taken together, illustrate the design space from classical (non–deep learning) segmentation to lightweight deep networks co-designed for FPGA deployment.

Sharma et al. (2016) describe a general methodology for mapping high-level deep neural models to FPGAs [52]. The work introduces an end-to-end flow and parameterizable hardware templates (for convolution, pooling, and fully connected layers), together with design-space exploration across tiling, parallelism, and precision. While not tailored specifically to segmentation, it established core practices—operator templating, dataflow composition, and quantization-aware mapping—that subsequent segmentation accelerators leverage.

Liu et al. (2020) present a fast and efficient FPGA accelerator for level set segmentation [53]. Rather than a CNN, the design accelerates the iterative PDE-based evolution of contours, exploiting fine-grained parallelism, pipelined updates, and on-chip memory to reduce iteration time. Although method-specific, it demonstrates how pixelwise iterative workloads can be mapped to deeply pipelined, stream-oriented hardware for segmentation, emphasizing low latency and energy efficiency.

Lin et al. (2022) propose a low–memory requirement MobileNet [54] accelerator on FPGA for auxiliary medical tasks [55]. The key idea is to exploit the structure of depthwise–separable convolutions to minimize on-chip buffer footprints and external memory bandwidth, aligning operator templates (DW/PW) with the hardware datapath. In practice, this means carefully scheduling line buffers, streaming weights/activations, and using compact precision to keep most traffic on chip—principles that translate directly to real-time segmentation backbones built from the same operator family.

Xu et al. (2024) target liver cancer segmentation with an FPGA-oriented lightweight deep inference pipeline [56]. The approach co-designs a compact network (relying on efficient blocks), reduced precision, and a streaming accelerator that keeps intermediate tensors on chip whenever possible. The result is a segmentation system tailored to medical imaging constraints (throughput, latency, and power) while maintaining clinically meaningful accuracy.

Collectively, these implementations support several consistent themes: (i) *operator-aware hardware* (e.g., depthwise/pointwise separation) to reduce memory and compute pressure; (ii) *dataflow pipelines* with line/First-In First-Out (FIFO) buffering; (iii) *quantization and compact precision* to fit bandwidth and on-chip memory budgets; and (iv) *parameterizable cores* to retarget different layers with the same building blocks. This body of evidence substantiates the design choices adopted in this thesis and indicates that lightweight, hardware-conscious segmentation pipelines on FPGAs are both practical and effective.

2.8 Quantization Methods for Deep Learning Models

Quantization is a technique used to reduce the precision of the numbers used to represent model parameters and activations, typically from 32-bit floating-point to lower bit-width formats such as 16-bit floating-point or 8-bit integer. This reduction in precision can lead to significant savings in memory usage and computational requirements, making it particularly beneficial for deploying models on resource-constrained devices like mobile phones and embedded systems.

Quantization can be applied in two ways:

- **Post-Training Quantization**: This method involves quantizing a pre-trained model without further training. It is straightforward but may lead to a drop in accuracy, especially for models that are sensitive to precision loss.
- **Quantization-Aware Training (QAT)**: In this approach, the model is trained with quantization in mind. During training, fake quantization operations simulate the effects of quantization, allowing the model to adapt and maintain higher accuracy when quantized.

Fixed-point representation is commonly used in quantization, where numbers are represented with a fixed number of bits for the integer and fractional parts. The fixed point value can be calculated using Equation 2.9:

$$\text{Fixed Point Value} = \text{Integer Value} \times 2^{-\text{Fractional Bits}} \tag{2.9}$$

where Integer Value is the quantized integer representation and Fractional Bits is the number of bits allocated for the fractional part. For example, an 8-bit fixed-point representation might allocate 4 bits for the integer part and 4 bits for the fractional part, allowing for a range of values from -8 to 7.9375 with a scale of 0.0625. The scale can be calculated using Equation 2.10:

$$\text{Scale} = 2^{-\text{Fractional Bits}} \tag{2.10}$$

This process reduces the number precision but also reduces the memory footprint and reduces the computational cost.

In summary, quantization is a powerful technique that utilizes fixed-point numbers representation for optimizing neural networks for deployment on edge devices, balancing the trade-off between model size, computational efficiency, and accuracy.

3 Model Exploration

Understanding how different models perform on medical imaging tasks is necessary for identifying the most suitable architecture to be implemented in FPGAs. This chapter presents an exploration of various neural network models, obtaining initial results and focusing on their structure, complexity, and suitability for resource constrained devices. This chapter also provides context for the selection of models evaluated in later chapters and highlights important considerations when choosing models and designing models.

3.1 Environment

In order to determine what network is better suitable for resource constrained devices a testing environment is necessary to ensure that experimental results are consistent and reproducible. The environment steps from data acquisition to model validation, as illustrated in Figure 3.1.

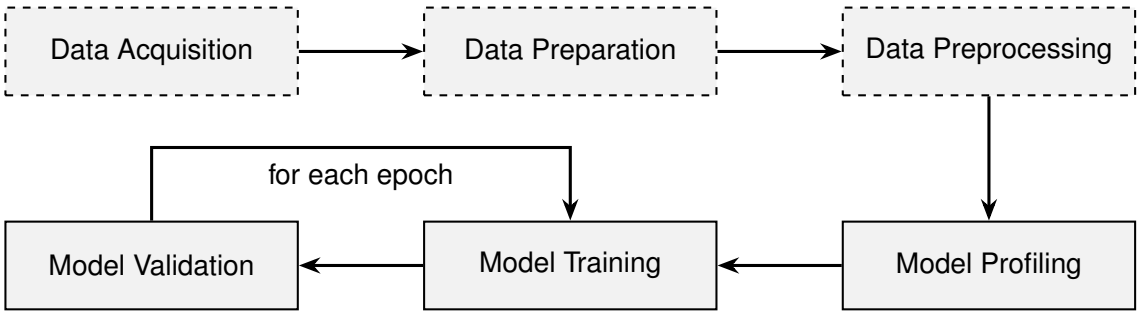


Figure 3.1: Environment pipeline

The environment begins with *Data Acquisition*, were publicly available medical imaging datasets relevant to the task were obtained. In the *Data Preparation* stage datasets are formatted into a suitable structure, splitting into training and validation. Next in the *Data Preprocessing* stage, training images are augmented, being resized to 256×256, randomly rotated 90, flipped, and normalized (per-channel standardization using Albumentations' `Normalize()`: $\mathbf{x} \leftarrow (\mathbf{x} - \boldsymbol{\mu}\, M)/(\boldsymbol{\sigma}\, M)$ with $M = 255$, where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the dataset mean and standard deviation computed on the training split).

Once the data pipeline is established, **model profiling** is conducted to evaluate each model's computational characteristics, specifically the total number of parameters and the number of multiply-accumulate operations (MACs). These metrics offer a quantitative measure of the model's complexity and potential computational cost. Following profiling, **model training** is carried out on identical hardware, with consistent optimization settings applied across all models,

including the learning rate, optimizer type, and batch size. During training, the model parameters are iteratively updated at each epoch. After each epoch, **model validation** is performed, during which each model is assessed using established quantitative metrics to monitor performance. The training and validation processes are executed sequentially within every epoch to ensure systematic evaluation and consistent progression throughout the learning process.

3.1.1 Data Preparation and Preprocessing

All datasets are organized under a uniform directory layout (see Section A.6), with images and masks stored in separate subdirectories: `<dataset>/{train,val}/{images,masks}`, where masks are stored per class in `masks/0`, `masks/1`, `...`, `masks/(C-1)`. Images and masks are selected by ID and file extension (`img_ext`, `mask_ext`) and, for multi-class problems, class-wise grayscale masks are stacked along the channel dimension.

If a dataset does not already provide a validation split, a train/validation split is created with a default validation fraction of `val_size = 0.3` (70/30 train/val). Datasets that already ship with explicit splits are loaded directly.

Images are resized to the configured input resolution (`input_h` $\times$ `input_w`) (see Section A.5). Training images are randomly rotated by 90 degrees (`RandomRotate90()`), flipped (`Flip()`), and resized (`Resize()`) to the desired network input, followed by normalization (`Normalize()`) from Albumentations. Validation images are resized (`Resize()`) and normalized (`Normalize()`).

The normalization (Equation 3.1) follows per-channel standardization where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the dataset mean and standard deviation computed on the training split. After transforms, images are converted to `float32` in the range $[0, 1]$ and laid out as CHW; masks are likewise converted to `float32` and stacked as C-channel tensors.

$$\mathbf{x} \leftarrow \frac{\mathbf{x} - \boldsymbol{\mu} \cdot M}{\boldsymbol{\sigma} \cdot M}, \quad M = 255, \tag{3.1}$$

3.1.2 Training

The training and validation environment was configured to obtain reproducible results. Special attention was taken for compatibility with the networks ensuring that the same environment configuration was used for each network. The configuration details are summarized in Table 3.1. The training code can be found in Appendix A.

During training, validation and inference steps, model performance is assessed using various metrics. Table 3.2 summarizes the metrics used throughout this document.

True Positive (TP) represent pixels correctly predicted as belonging to the target class (e.g., lesion pixels correctly segmented). True Negative (TN) are pixels correctly identified as background. False Positive (FP) occur when background pixels are mistakenly labeled as part of the target object, causing over-segmentation. False Negative (FN) are pixels that truly belong to the target but are missed by the model, causing under-segmentation A represents the predicted region and B represents the ground truth region.

Hardware	NVIDIA GeForce RTX 4080 Driver Version: 535.183.01
Framework	PyTorch 2.1.0 with CUDA 11.8
Training Conditions	Epochs: 300; Batch size: 8; Seed: 41
Dataset Split	Training: 70%; Validation: 30%
Input Image Size	256×256 pixels
Optimizer	Adam; Learning rate: 1×10^{-3}; Weight decay: 1×10^{-4}
LR Scheduler	Cosine Annealing; T_{max}: 1×10^{-3}; η_{min}: 1.0×10^{-5}

Table 3.1: Environment configuration.

Metric	Description	Formula
Accuracy	Ratio of correctly classified pixels	$\frac{TP+TN}{TP+TN+FP+FN}$
Sensitivity (Recall)	Ratio of true positives detected	$\frac{TP}{TP+FN}$
Precision	Ratio of correct positive predictions	$\frac{TP}{TP+FP}$
F1-Score	Harmonic mean of precision and sensitivity	$\frac{2 \cdot \text{Precision} \cdot \text{Sensitivity}}{\text{Precision} + \text{Sensitivity}}$
Intersection over Union (IoU)	Overlap between predicted and ground truth regions	$\frac{\lvert A \cap B \rvert}{\lvert A \cup B \rvert}$
Dice Coefficient	Segmentation overlap measure (similar to F1-score for sets)	$\frac{2\lvert A \cap B \rvert}{\lvert A \rvert + \lvert B \rvert}$
Foreground Accuracy	Fraction of correctly predicted foreground pixels, ignoring true negatives (for inference only)	$\frac{TP}{TP+FP+FN}$

Table 3.2: Metrics used for model performance assessment

Training *Loss Function* uses a combined Binary Cross-Entropy (BCE) and Dice Loss:

$$\mathcal{L} = 0.5 \cdot \text{BCE} + \text{Dice Loss} \tag{3.2}$$

The BCE component penalizes pixel-wise classification errors, while the Dice Loss component encourages overlap between predicted and ground truth masks, this combination helps balance pixel accuracy with region-level agreement making it a suitable loss functions in segmentation tasks. This is particularly important in medical imaging, where the target structures may occupy a small fraction of the image.

It is important to note that for binary segmentation tasks such as those studied in this work, the overall pixel accuracy is not an ideal performance indicator. This is because most of the pixels in a medical image belong to the background class (black), and a model that simply predicts all pixels as background achieves very high accuracy despite being completely useless for segmentation. To better capture performance in these scenarios, the assessment relies

on overlap-based and class-specific metrics such as Intersection over Intersection over Union (IoU) and F1.

3.2 Inference

Once a model is trained, the goal is to perform inference on unseen data. The inference stage takes as input a trained model checkpoint and a test image, and produces the corresponding segmentation mask. Section A.4 provides the complete Python script used for this task.

The script begins by loading the configuration and weights of the trained model, ensuring that preprocessing (resizing, normalization, tensor formatting) matches the settings used during training. It then runs a forward pass of the model on the input image under `torch.inference_mode()`. If a ground truth mask is available, the script computes a foreground accuracy (F-Accuracy) metric, which evaluates the alignment between predicted and target masks. This metric is better suitable for assessing segmentation quality than the basic accuracy metric in the binary predictions, as it reflects how well the model captures the relevant structures without being dominated by the background pixels. Finally, the predicted segmentation is binarized, and written to disk as a grayscale image.

In addition to single-image inference, the code supports a recursive mode, which automatically traverses experiment directories, finds the most recent checkpoint, and runs inference for each dataset. This allows for rapid evaluation across multiple trained variants without manual intervention.

3.3 Benchmarking Networks

Several medial imaging segmentation networks were selected to evaluate. Table 3.3 lists the networks used in the experiments along with their MACs and parameter counts. Fewer MACs generally indicate faster models, which are more suitable for devices with limited resources.

Network	MACS	PARAMS
AttentionUNet [40]	66.632G	34.879M
TransUnet [41]	32.229G	105.322M
UNet [30]	65.522G	34.527M
UNet++ [39]	34.903G	9.160M
UNet_V2 [11]	5.098G	24.903M
CFPNetM [14]	3.472G	758.881K
GIVTED-Net [45]	374.276M	192.193K
LUCF-Net [48]	8.590G	6.931M
ULite [44]	756.957M	878.417K
UNeXt [57]	573.022M	1.472M
CMUNeXt [49]	7.418G	3.149M

Table 3.3: Networks used in the experiments.

The networks selected cover a range of architectures, from traditional encoder-decoder designs (UNet, UNet++, AttentionUNet) to transformer based models (TransUNet) and recent lightweight architectures (CFPNetM, GIVTED-Net, LUCF-Net, ULite, UNeXt, CMUNeXt). This diversity allows for the evaluation of different designs and their trade-offs between accuracy and efficiency in the context of medical image segmentation. Also, these networks have publicly available PyTorch implementations facilitating reproducibility and comparison.

3.4 Datasets for Medical Image Segmentation

The datasets chosen for this study cover a wide range of medical imaging each presenting unique challenges. These datasets are very common for medical imaging segmentation and are present in multiple articles results. The datasets used are: Breast UltraSound Image (BUSI) [58], International Skin Imaging Collaboration (ISIC) 2016 [59], and Fundus Image Dataset for Artificial Intelligence based Vessel Segmentation (FIVES) [60].

3.4.1 BUSI

The BUSI [58] public dataset collected from 600 female patients, includes 780 breast ultrasound images, covering 133 normal cases, 487 benign cases, and 210 malignant cases, each with corresponding ground truth labels. Only the benign and malignant cases from this dataset are utilized. Figure 3.2 illustrates the data contained in the BUSI dataset.

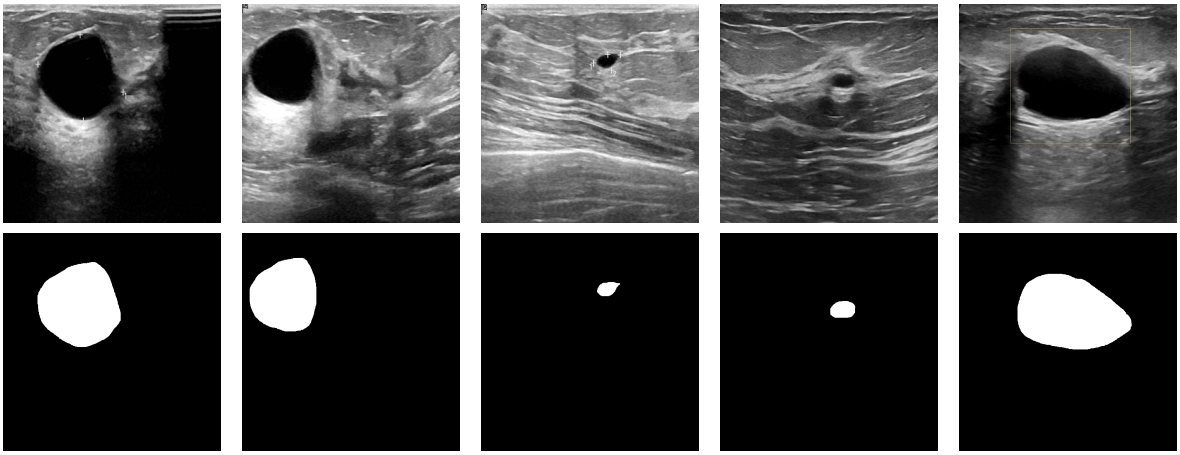


Figure 3.2: Top: Input ultrasound images. Bottom: Corresponding segmentation masks. Copyright 2020, Elsevier [58].

3.4.2 ISIC (2016)

The ISIC 2016 [59] challenge dataset consists of dermoscopic images for skin lesion classification. It includes 900 training images and 379 test. Each image is provided with corresponding ground truth segmentation masks and diagnostic labels. Figure 3.3 illustrates the data contained in the ISIC dataset.

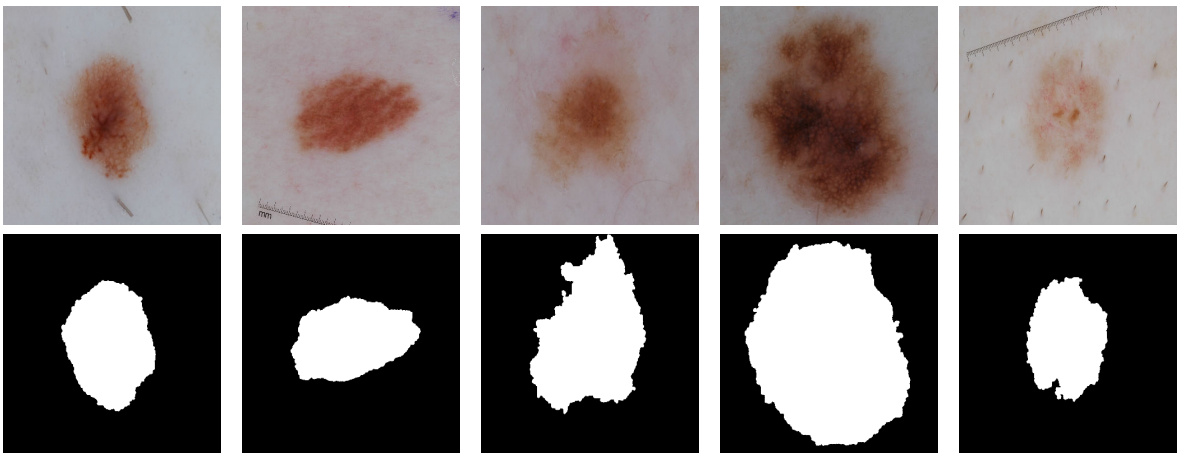


Figure 3.3: Top: Input dermoscopic images. Bottom: Corresponding segmentation masks. Copyright 2016, The International Skin Imaging Collaboration [59].

3.4.3 FIVES

Retinal vasculature provides an opportunity for direct observation of vessel morphology, which is linked to multiple clinical conditions. However, objective and quantitative interpretation of the retinal vasculature relies on precise vessel segmentation, which is time-consuming and labor-intensive [61]. FIVES [60] dataset consists of 800 high-resolution multi-disease color fundus photographs with pixel-wise manual annotation. The annotation process was standardized through crowdsourcing of a group of medical experts. The quality of each image was evaluated, including illumination and color distortion, blur, and low contrast distortion, based on which the data splitting was conducted to make sure the balanced distribution of image features. Figure 3.4 illustrates the data contained in the FIVES dataset.

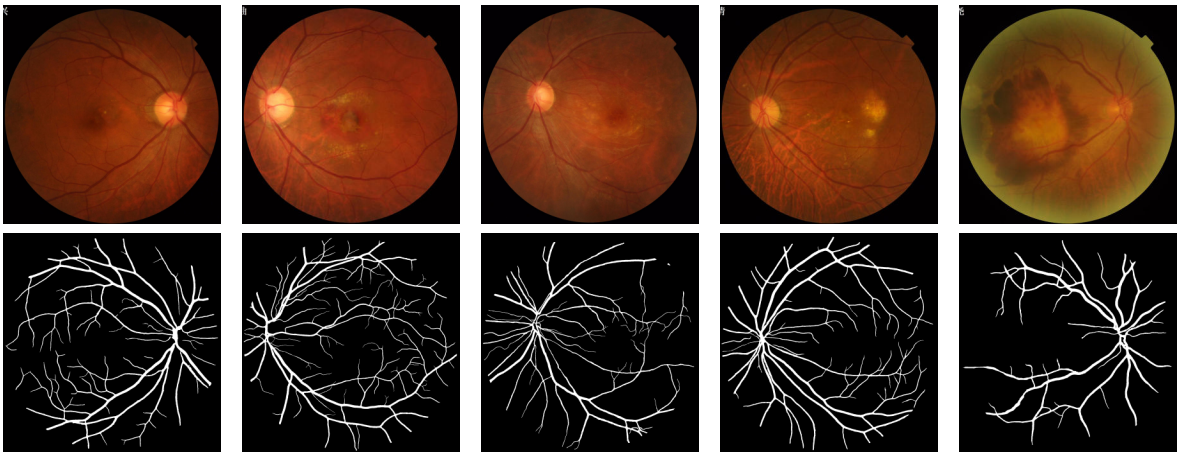


Figure 3.4: Top: Input fundus images. Bottom: Corresponding segmentation masks. Copyright 2022, Nature Publishing Group UK London [60].

3.5 Initial Benchmark

An initial benchmarking was performed to obtain base results for trade-off between model performance and resource usage. The selected models vary significantly in terms of computational complexity and parameter count, key factors in determining the effective deployment on

resource-constrained hardware.

Table 3.4 summarizes the validation results for each model, F1-score and IoU are used as primary performance metrics, alongside MACs and parameter count. Higher values of F1 and IoU indicate better segmentation quality, while lower MACs and parameter counts reflect lower computational demand. Results are split into conventional encoder-decoder models, transformer-based models, and recent lightweight architectures.

Network	MACs (G)↓	Params (M)↓	BUSI		FIVES		ISIC2016	
			F1↑	IoU↑	F1↑	IoU↑	F1↑	IoU↑
AttentionUNet	66.63	34.88	74.53	61.9	82.08	69.75	90.76	83.4
UNet	65.52	34.53	74.89	61.1	80.18	67.0	90.59	83.0
UNet++	34.9	9.16	73.88	61.16	**89.2**	**80.54**	90.88	83.67
UNet_V2	5.1	24.9	76.37	63.28	72.94	57.43	90.94	83.73
TransUnet	32.23	105.32	77.81	65.2	87.85	78.37	91.56	84.69
CFPNetM	3.47	0.76	**79.76**	**67.1**	<u>88.3</u>	<u>79.06</u>	91.88	85.16
CMUNeXt	7.42	3.15	78.94	65.84	81.46	68.77	91.79	85.0
GIVTED-Net	**0.37**	<u>0.19</u>	77.92	64.6	86.37	76.02	**92.34**	**85.91**
LUCF-Net	8.59	6.93	<u>79.1</u>	<u>66.17</u>	76.51	62.03	<u>92.04</u>	<u>85.37</u>
ULite	0.76	0.88	78.93	66.11	77.58	63.44	91.49	84.47
UNeXt	0.57	1.47	77.56	64.01	59.72	42.71	91.32	84.18

Table 3.4: Initial validation results. Best results are highlighted in bold, second-best are underlined

The results in Table 3.4 reveal a clear accuracy–efficiency trade-off. Conventional encoder–decoders (UNet, AttentionUNet) are competitive on ISIC2016 (F1 $\approx$ 90.6–90.8) but come with very high cost (65–67 G MACs, 34–35 M params). UNet++ moderates this burden (34.9 G MACs, 9.16 M params) while delivering the best FIVES score (F1 89.2 / IoU 80.54), showing that architectural refinement within this family can meaningfully improve the balance. Transformer-based TransUNet achieves strong accuracy (e.g., BUSI F1 77.81; ISIC2016 F1 91.56) but with the highest complexity by far (32.23 G MACs, 105.3 M params), making it a poor fit for hardware-constrained devices.

In contrast, lightweight networks dominate in the accuracy–efficiency category. CFPNetM (3.47 G MACs, 0.76 M params) ranks first on BUSI (F1 79.76 / IoU 67.10) and second on FIVES (F1 88.30 / IoU 79.06) at near-minimal cost. GIVTED-Net pushes compute and parameters even lower (0.37 G MACs, 0.19 M params) while topping ISIC2016 (F1 92.34 / IoU 85.91). LUCF-Net maintains high accuracy (BUSI F1 79.10; ISIC2016 F1 92.04) at moderate cost (8.59 G MACs, 6.93 M params). ULite (0.76 G MACs, 0.88 M params) delivers a strong BUSI score (F1 78.93) with sub-1 M parameters, underscoring how far compactness can go with limited accuracy loss. UNeXt is extremely efficient (0.57 G MACs, 1.47 M params) but shows dataset sensitivity on FIVES (F1 59.72). CMUNeXt, built around depthwise–separable operators, strikes a solid balance (BUSI F1 78.94; ISIC2016 F1 91.79) at moderate cost (7.42 G MACs, 3.15 M params).

Overall, lightweight models consistently deliver near-State of the Art (SoTA) performance at one

to two orders of magnitude lower cost, making them strong candidates for FPGA deployment. Larger encoder–decoder and transformer models, although accurate, carry computational footprints that are prohibitive under strict hardware constraints. Finally, dataset variability matters: several compact networks generalize well across BUSI and ISIC2016, while FIVES exhibits a wider spread—suggesting that lightweight designs are excellent for targeted, relatively homogeneous medical datasets, whereas higher-capacity models may be better suited to more generic, heterogeneous, or multi-class scenarios.

3.6 Network Selection

The selection of neural network models for deployment on FPGAs depends on multiple factors, not only the initial benchmark performance but also the computational efficiency and practical feasibility (effective implementation) of the network. The general criteria used to identify the possible model candidates is as follows:

- **Performance, Profiling and Architectural Analysis**: Candidate models were assessed based on their segmentation performance. Characteristics such as the number of parameters and the total number of MACs directly influence the computational and memory requirements. Ultimately the selection should prioritize models with greater trade-off between accuracy and efficiency.
- **Tolerable Performance Margin**: Given the trade-off between model complexity and segmentation accuracy, the maximum allowable performance degradation is of 10%—relative to the best-performing model in that dataset. Models within this margin should be selected if they offer substantial gains in terms of resource efficiency (derived from model profiling and architectural analysis), critical for realtime processing.
- **Reproducibility and Implementation Quality**: Deployment also requires that models are well-documented and with reproducible implementations. Preference was given to models with open-source repositories, complete training and validation pipelines, and clearly described architectural details.

Based on the selection criteria outlined — balancing segmentation performance, computational efficiency, and implementation feasibility — the selected models are characterized in Table 3.5.

In order to narrow down the candidates to a singular model some more objective criteria were imposed, assessing the suitability of the candidates for deployment on FPGAs, Table 3.6 evaluates the selected candidates following some more specific criteria.

Given the analysis in Table 3.6, **CMUNeXt** was selected as the model of choice for this project, the model offers an optimal balance between segmentation accuracy, computational efficiency, and effective deployment. CMUNeXt stands out from the other candidates as it provides a highly modular design, which allows for architectural modifications, facilitating the integration into custom hardware. The model demonstrates consistently competitive results across datasets, while maintaining a lightweight structure in terms of parameter count and MACs.

Model	Year	Params (M)	MACs (G)	Summary
CFPNetM	2023	0.76	3.47	Lightweight model using multiscale attention. Strong accuracy on BUSI and FIVES with minimal resources.
CMUNeXt	2024	3.15	7.42	Modular hybrid CNN architecture. Balances spatial detail and context with good reproducibility.
GIVTED-Net	2024	0.19	0.37	Efficient design that combines global attention and inverse-local context. High accuracy with lowest compute cost.

Table 3.5: Comparison of selected model based on parameters and MACs, and a summary of their architectural characteristics

Criteria	CFPNet-M	CMUNeXt	GIVTED-Net
Top-tier Performance	✓	✓	✓
Low Computational Cost	✓	✓	✓
Reproducibility	✓	✓	✓
Modular Design	✓	✓	≈
Implementation Simplicity	✓	✓	≈
Documentation Quality	✓	✓	✓
Open-Source Article	✓	✓	✗
Available Code	✓	✓	✓

Table 3.6: Comparative analysis of candidate models across detailed selection criteria. Green checkmarks (✓) indicate that the model meets the criterion. Red crosses (✗) indicate that the model does not meet the criterion. Orange approximate (≈) represent partial or moderate satisfaction. Underlined green checkmarks (✓) highlight the strongest candidate for that specific criterion.

Additionally, the availability of high-quality documentation and reproducible code significantly reduces the overhead associated with implementation and validation.

3.7 Final Network Candidate

CMUNeXt poses as well-suited for further optimization and hardware acceleration implementation. The next sections will delve into the steps taken to simplify and optimize this network for hardware deployment.

The CMUNeXt network has a similar architectural design to the one described by U-Net, but with some noticeable improvements to retain more information. These improvements come in the form of: stem blocks, CMUNeXt blocks with depthwise separable convolutions and Skip fusion blocks where features are mixed. Figure 3.5 describes the CMUNeXt architecture in detail.

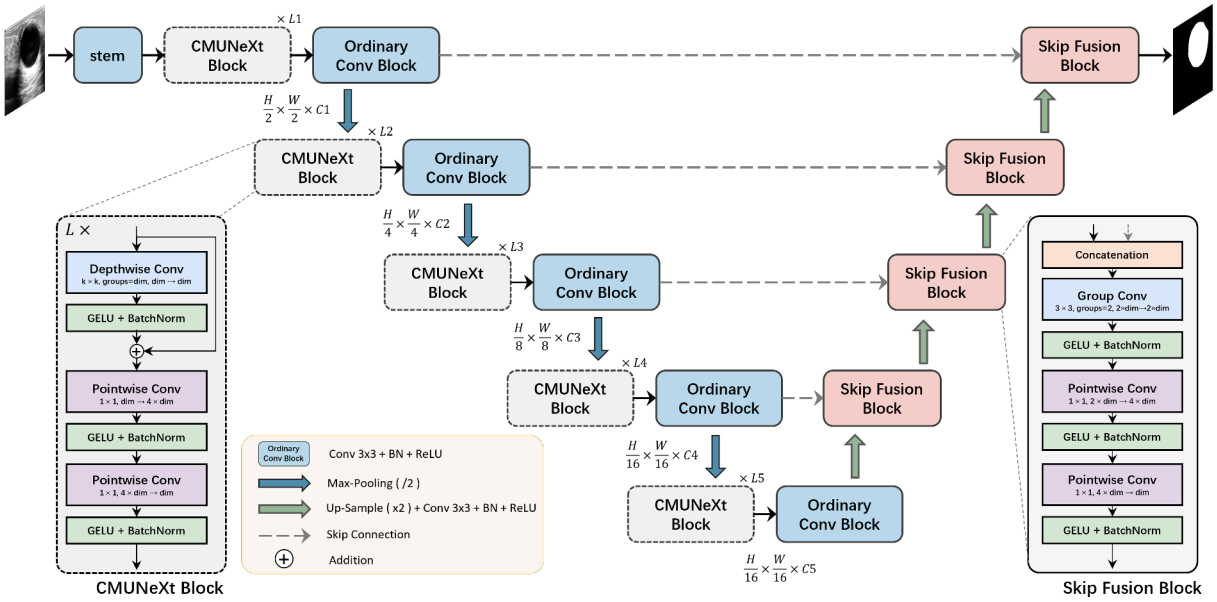


Figure 3.5: CMUNeXt Architecture

Stem The stem block serves as the initial feature extractor. It uses a 3×3 convolution with stride 1 and padding 1, followed by batch normalization and an in-place GeLU activation, helping preserve the spatial resolution while transforming the raw image input into a feature-rich representation for subsequent processing.

CMUNeXt Block This is the core building block of the encoder, based on depthwise separable convolutions for computational efficiency. Each block includes one depthwise convolution, followed by an expansion via a pointwise convolution (increasing the channel dimension by a factor of 4), and then a contraction back to the original dimension. All convolutions are followed by batch normalization and GeLU activations. These blocks are repeated multiple times to progressively extract hierarchical features.

Bottleneck The bottleneck sits at the interface between the deepest encoder stage and the start of the decoder—i.e., after stage $L5$ and before the first upsampling step. Because the encoder applies four 2×2 poolings, the feature map at this point has the smallest spatial support in the network: for an input of size $H\times W$, it is reduced to $\frac{H}{16}\times\frac{W}{16}$.

Downsampling Downsampling is performed using 2×2 max pooling with stride 2, reducing the spatial resolution by half while retaining important features. This allows the network to build deeper representations with manageable computational cost.

Upsampling Upsampling in the decoder is done using bilinear interpolation to double the spatial resolution, followed by a 3×3 convolution, batch normalization, and GeLU activation. This helps refine and reconstruct higher-resolution feature maps from the compressed encoder output.

Skip Fusion The Skip Fusion block combines encoder features with decoder outputs to preserve spatial detail. It uses grouped convolutions to separately process encoder and decoder inputs, then fuses them through pointwise convolutions. Each step is followed by batch normalization and GeLU. This block enhances the network's ability to recover fine-grained details lost during downsampling.

The architecture can be customized with several parameters: the number of channels at each depth (C), the length of CMUNeXt blocks per depth (L), and the kernel size used at each depth (K). The network is structured with 5 hierarchical depths, in each depth the input image dimensions are divided by 2 when encoding and multiplied by 2 when decoding. Each depth is configured with distinct values for these parameters.

Given the flexibility of the architecture, it can be tailored to meet specific performance and resource requirements. This is particularly important when considering deployment on resource-constrained hardware, where trade-offs between accuracy and efficiency must be carefully managed. In the next chapter, the focus will be on optimizing CMUNeXt for FPGA deployment to further reduce its computational footprint while still maintaining acceptable segmentation performance.

4 Mobile-CMUNeXt

Mobile-CMUNeXt represents the culmination of architectural optimizations applied to the CMUNeXt network, designed specifically for deployment in FPGAs. This chapter details the design objectives, the code-level changes achieved through a combination of channel pruning, activation function simplification, skip connection restructuring, and layer reordering, and the empirical results that validate the approach. All code referenced in this chapter is available in Section B.2.

4.1 Optimizing CMUNeXt

This section presents a series of architectural and implementation-level optimizations applied to the original CMUNeXt model. The goal of these enhancements is to improve inference speed, computational efficiency, and facilitate FPGA deployment. Each proposed modification targets a particular aspect of the network — such as feature representation and depthwise operations — while preserving the lightweight nature of the original design. The following subsections describe these optimizations in detail, along with their impact on performance.

4.1.1 Configuration Parameters

The original CMUNeXt provides multiple configurations that trade off model size against accuracy by varying channel widths, block lengths, and kernel sizes. An overview appears in Table 4.1, with performance in Table 4.2.

Network	Number of Channels					Length of Blocks					Kernel Size				
	$C1$	$C2$	$C3$	$C4$	$C5$	$L1$	$L2$	$L3$	$L4$	$L5$	$K1$	$K2$	$K3$	$K4$	$K5$
CMUNeXt-L	32	64	128	256	512	1	1	1	6	3	3	3	7	7	7
CMUNeXt	16	32	128	160	256	1	1	1	3	1	3	3	7	7	7
CMUNeXt-S	8	16	32	64	128	1	1	1	1	1	3	3	7	7	9

Table 4.1: CMUNeXt configurations with varying parameters.

To increase computational efficiency, model complexity is reduced through configuration optimization. The proposed *Mobile-CMUNeXt* variant is an ultra-lightweight instance of the architecture. Its configuration parameters — channel dimensions, block lengths, and kernels — are summarized in Table 4.3. Compared to standard variants, channels are substantially reduced; for example, the deepest stage uses only 24 channels versus 512 in CMUNeXt-L.

Network	MACs (G)↓	Params (M)↓	BUSI		FIVES		ISIC2016	
			IoU↑	F1↑	IoU↑	F1↑	IoU↑	F1↑
CMUNeXt-L	17.183G	8.287M	78.91	66.16	84.03	72.51	91.92	85.2
CMUNeXt	7.418G	3.149M	77.64	64.42	81.52	68.87	91.72	84.88
CMUNeXt-S	1.090G	417.505K	78.49	65.19	80.47	67.4	91.52	84.48

Table 4.2: Validation results for different CMUNeXt variants.

A simple $8\times$ rule is adopted: stage widths are small and constrained to multiples of eight ($C \in \{8, 16, 24, \dots\}$). This configuration was chosen specifically for the FPGA implementation, in order to process 8 channels at a time.

Also, the number of blocks in stages *L*1, *L*4, and *L*5 are increased to retain feature extraction capacity with minimal parameter growth. This change in block length finds a good trade-off between MACs and parameters: the additional blocks in *L*1 enhance the early stages of the model, where the channel count is very low (8), and the increases in *L*4 and *L*5 help preserve higher-level features—similar to the approach used in the CMUNeXt-L variant.

Network	Number of Channels					Length of Blocks					Kernel Size				
	*C*1	*C*2	*C*3	*C*4	*C*5	*L*1	*L*2	*L*3	*L*4	*L*5	*K*1	*K*2	*K*3	*K*4	*K*5
CMUNeXt-L	32	64	128	256	512	1	1	1	6	3	3	3	7	7	7
CMUNeXt	16	32	128	160	256	1	1	1	3	1	3	3	7	7	7
CMUNeXt-S	8	16	32	64	128	1	1	1	1	1	3	3	7	7	9
Mobile-CMUNeXt	8	8	16	16	24	3	1	1	2	3	3	3	7	7	9

Table 4.3: CMUNeXt configurations and proposed *Mobile-CMUNeXt* configuration.

The *Mobile-CMUNeXt* configuration requires only 486 million MACs, compared to 17 billion MACs in CMUNeXt-L — a reduction of approximately 97.3%. Parameter count is reduced from 8 million to 47 thousand (99.4% smaller), as shown in Table 4.4.

Network	MACS	PARAMS
CMUNeXt-L	17.183G	8.287M
CMUNeXt	7.418G	3.149M
CMUNeXt-S	1.090G	417.505K
Mobile-CMUNeXt	**486.621M**	**46.569K**

Table 4.4: CMUNeXt parameters and MACs comparison.

A first training cycle using this configuration yields the following validation results for the *Mobile-CMUNeXt* configuration:

- **BUSI:** F1 = 0.78, IoU = 0.65
- **FIVES:** F1 = 0.82, IoU = 0.69

- **ISIC (2016):** F1 = 0.95, IoU = 0.84

In comparison, the *Mobile-CMUNeXt* configuration achieves an average reduction of more than two orders of magnitude in computational and memory cost while maintaining accuracy within 1–3% of the larger CMUNeXt-S model across datasets. On BUSI and FIVES, its F1 and IoU scores remain competitive (0.78/0.65 and 0.82/0.69, respectively), while on ISIC (2016) it nearly matches the best-performing variants with an F1 of 0.95 and IoU of 0.84. This indicates that the architecture retains strong feature extraction capability despite its extreme compression.

The study confirms that the *Mobile-CMUNeXt* configuration achieves a substantial reduction in both computational cost and parameter count while retaining competitive segmentation performance. Despite being over 97% lighter than CMUNeXt-L, the model maintains strong accuracy across all datasets. This balance between compactness and accuracy makes the *Mobile-CMUNeXt* configuration suitable for deployment on resource-constrained hardware platforms.

4.1.2 Activation Function

In the original CMUNeXt architecture, the GELU Equation 4.1 activation function is used due to its smooth and expressive non-linearity. However, GELU is very computationally intensive making it not ideal the FPGA deployment scenario.

GELU is defined as:

$$\text{GELU}(x) = x \cdot \Phi(x) \tag{4.1}$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution.

To reduce computational overhead without significantly impacting performance an experiment with multiple activation functions was performed in order to identify a simpler alternative that retains competitive accuracy while improving efficiency and compatibility with quantization. The candidate functions evaluated include ReLU, LeakyReLU, Hardswish, and GELU itself. Each function was tested under identical conditions across training runs, with results summarized in Table 4.5.

One of the initial alternatives considered was Hardswish Equation 4.2 — popularized in MobileNetV3 [62]. Hardswish approximates the shape of GELU using a piecewise linear function, offering a good balance between non-linearity and computational efficiency. Its formula is:

$$\text{Hardswish}(x) = \begin{cases} 0, & \text{if } x \leq -3, \\ x, & \text{if } x \geq +3, \\ x \cdot (x+3) \cdot \frac{1}{6}, & \text{otherwise.} \end{cases} \tag{4.2}$$

Figure 4.1 compares Hardswish, GELU, ReLU and the LeakyReLU (with a negative slope of 0.1), visually showing how Hardswish approximates GELU's smoothness while ReLU and Leaky ReLU exhibit sharper transitions but lower computational complexity.

Despite the theoretical advantages of Hardswish, **ReLU** was selected for deployment due to its hardware-friendliness on FPGAs. In contrast to GELU or Hardswish, ReLU reduces to a

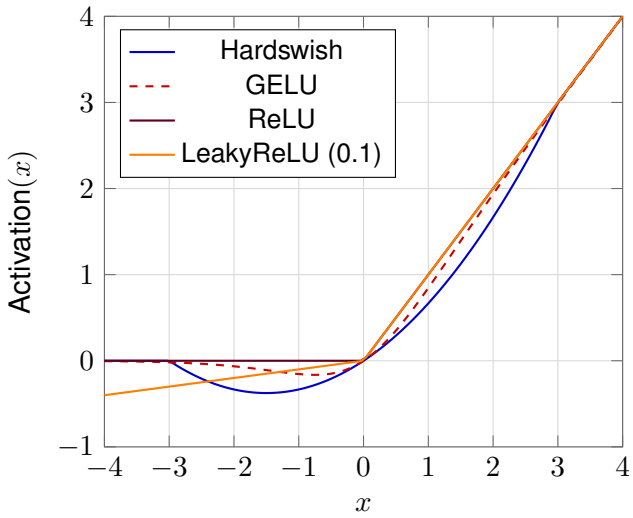


Figure 4.1: Comparison of activation functions.

simple $\max(0, x)$ operation, avoiding costly computations. Also, it integrates with low-precision quantization (integer activations with batch-normalization merge Section 5.2). Overall **ReLU** accuracy on the target datasets is comparable to other activations (see Table 4.5), providing a favorable trade-off between accuracy, throughput, and implementation simplicity for the FPGA target.

Looking at the results in Table 4.5, performance differences between activations are relatively small across datasets. GELU and Hardswish are often slightly better on BUSI and FIVES, while ReLU and LeakyReLU match or even surpass them on ISIC2016. For instance, in the CMUNeXt variant, ReLU achieves the highest IoU (85.48) on ISIC2016, while remaining highly competitive on BUSI and FIVES. Importantly, the accuracy gap between ReLU and more complex functions is typically less than 1–2 points, which is negligible compared to the hardware costs saved.

Despite the theoretical advantages of GELU and Hardswish, **ReLU** is the most suitable option for hardware deployment. ReLU simplifies to a $\max(0, x)$ operation, avoiding costly computations associated with GELU and Hardswish. This simplicity translates into reduced resource usage and improved latency on FPGAs. Furthermore, ReLU integrates naturally with quantization pipelines, particularly when batch normalization (Section 5.2) is merged into preceding layers, ensuring stable integer-based inference. For these reasons, ReLU provides a favorable balance between accuracy, throughput, and hardware efficiency, making it a valid choice for FPGA deployment.

4.1.3 Reordering Layers

In the original CMUNeXt design, both the *CMUNeXt ConvBlock* and *Skip Fusion* modules used the following order of operations: *Convolution* → *Activation* → *Batch Normalization (BN)*. This was updated to the more standard sequence: *Convolution* → *Batch Normalization* → *Activation*. The code change is reflected in Listing 4.1.

Network	Activation	BUSI		FIVES		ISIC2016	
		F1↑	IoU↑	F1↑	IoU↑	F1↑	IoU↑
CMUNeXt-S	GELU	**78.49**	**65.19**	80.47	67.40	91.52	84.48
	Hardswish	77.41	64.06	82.62	70.46	91.45	84.40
	LeakyReLU	75.51	62.96	81.21	69.33	91.72	84.82
	ReLU	76.60	63.05	**82.80**	**70.71**	**91.76**	**84.94**
CMUNeXt	GELU	77.64	64.42	81.52	68.87	91.72	84.88
	Hardswish	78.87	66.0	**82.12**	**69.73**	91.98	85.34
	LeakyReLU	**79.71**	**66.92**	74.94	60.13	91.72	84.80
	ReLU	79.12	66.03	81.02	68.2	**92.08**	**85.48**
CMUNeXt-L	GELU	**78.91**	**66.16**	84.03	72.51	91.92	85.20
	Hardswish	78.90	65.99	84.99	73.92	**91.89**	**85.15**
	LeakyRELU	78.47	65.47	**87.09**	**78.73**	91.22	84.04
	RELU	78.67	65.61	86.11	75.64	91.38	84.31

Table 4.5: Comparison of different activation functions evaluated for CMUNeXt.

```
class ConvBlock(nn.Module):
    def __init__(self, ch_in, ch_out, act: nn.Module = nn.ReLU):
        super().__init__()
        self.conv = nn.Sequential(
-           nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1, bias=True),
-           act(inplace=True),
-           nn.BatchNorm2d(ch_out),
+           nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1, bias=True),
+           nn.BatchNorm2d(ch_out),
+           act(inplace=True),
        )
```

Listing 4.1: Replace Conv → Act → BN with Conv → BN → Act for ConvBlock

This change improves compatibility with common optimization techniques such as weight initialization and pruning. More importantly it enables *batch normalization merging* (Section 5.2), fusing the batch normalization with convolution to reduce runtime operations.

4.1.4 Depthwise Separable Convolutions

To counter the reduced parameters and subsequent reduced information retention (Section 4.1.1), CMUNeXt blocks were extended by introducing an additional depthwise convolution. This modification maintains the model's lightweight nature while increasing the capacity to retain spatial information.

The aim is to enhance the non-linear representational power without significantly increasing the parameter count by stacking two residual depthwise convolutions before the expansion and projection stages. The PyTorch implementation of this change is shown in Listing 4.2.

```
class MobileCMUNeXtBlock(nn.Module):
    def __init__(self, ch_in, ch_out, depth=1, k=3, act: nn.Module = nn.ReLU):
```

```
        super().__init__()
        self.block = nn.Sequential(
            *[
                nn.Sequential(
                    # First depthwise conv (residual)
                    Residual(
                        nn.Sequential(
                            nn.Conv2d(ch_in, ch_in, kernel_size=(k, k), groups=ch_in,
                                padding=(k//2, k//2)),
                            nn.BatchNorm2d(ch_in),
                            act(),
                        )
                    ),
-                   # (originally: go straight to expand/project here)
+                   # Second depthwise conv (new) (residual)
+                   Residual(
+                       nn.Sequential(
+                           nn.Conv2d(ch_in, ch_in, kernel_size=(k, k), groups=ch_in,
    padding=(k//2, k//2)),
+                           nn.BatchNorm2d(ch_in),
+                           act(),
+                       )
+                   ),
                    # 1x1 expand -> 1x1 project
                    nn.Conv2d(ch_in, ch_in * 4, kernel_size=1),
                    nn.BatchNorm2d(ch_in * 4),
                    act(),
                    nn.Conv2d(ch_in * 4, ch_in, kernel_size=1),
                    nn.BatchNorm2d(ch_in),
                    act(),
                )
                for _ in range(depth)
            ]
        )
```

Listing 4.2: MobileCMUNeXtBlock: one → two residual depthwise convolutions

Listing 4.2 shows the PyTorch implementation. The class `MobileCMUNeXtBlock` defines the adapted CMUNeXt blocks and builds its body as a repeated (`depth`-times), the layers sequence is as it follows:

- **Residual depthwise #1.** The first sub-block is wrapped in `Residual(...)` to add a skip connection around a depthwise convolution `nn.Conv2d(ch_in, ch_in, kernel_size=(k,k), groups=ch_in, padding=k//2)` followed by `BatchNorm2d` and the chosen activation `act()`. Setting `groups=ch_in` enforces one filter per channel (true depthwise), and `padding=k//2` keeps $H \times W$ unchanged.
- **Residual depthwise #2 (new).** A second, identical residual depthwise block is inserted immediately after. This is the only structural change compared to the original CMUNeXt block, and it increases the network's ability to model spatial patterns without altering the tensor shape.
- **Expand–project (1×1).** The sequence `Conv2d(ch_in, 4*ch_in, 1)` → `BN` → `act()` → `Conv2d(4*ch_in, ch_in, 1)` → `BN` → `act()` performs channel mixing. The first 1×1 "expands" channels by $\times4$ to increase intermediate capacity; the second "projects" back to `ch_in`, restoring the interface expected by the outer residual path.

Figure 4.2 illustrates the revised architecture of the *CMUNeXt Block*, (now called *Mobile-CMUNeXt Block*) where the two depthwise convolutions are applied in sequence — each within a residual connection — followed by an expanding pointwise convolution ($C \times 4$) and a contracting pointwise convolution. This block depth is repeated according to the configuration specified in Table 4.3. Finally, a standard convolutional block (*ConvBlock*) is applied after each stage to further refine features.

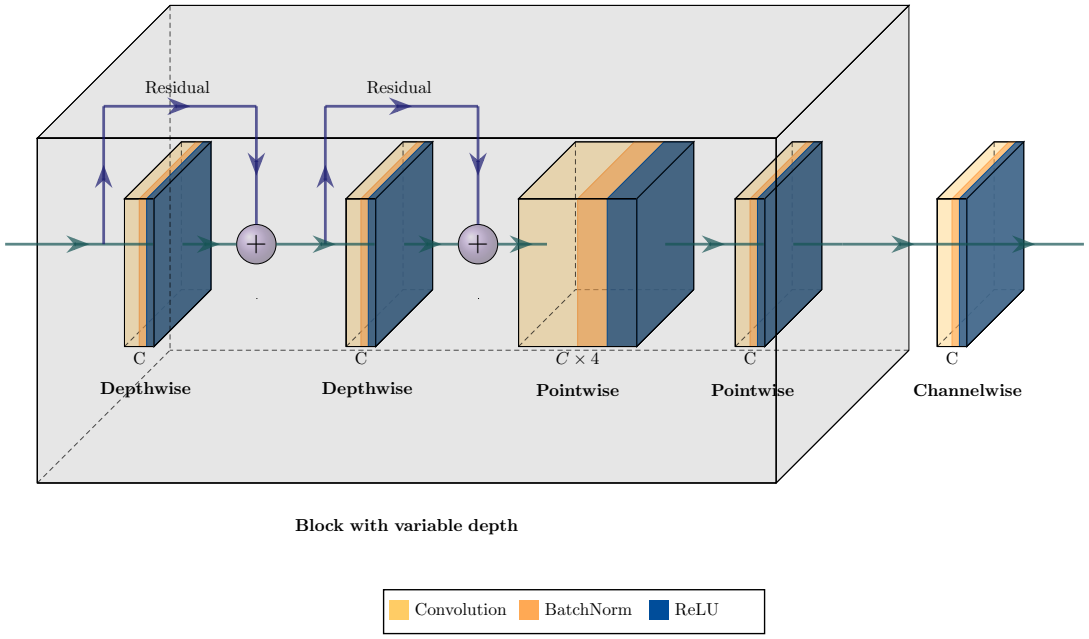


Figure 4.2: Modified CMUNeXt block with two depthwise separable convolutions in sequence, used to enhance spatial representation while keeping the model lightweight.

4.1.5 Skip Connections

In architectures such as U-Net skip connections are implemented using concatenation operations between encoder and decoder feature maps. While this approach preserves more feature information by maintaining separate channels for both paths it also significantly increases memory usage.

In the original CMUNeXt design skip connections concatenate encoder and decoder feature maps with a maximum shape of $256 \times 256 \times 8$, resulting in intermediate tensors of size $256 \times 256 \times 16$. This doubling of channels leads to higher memory consumption and increased computational cost in subsequent layers.

Replacing these concatenation-based skip connections defined in the *Skip Fusion* block with element-wise addition (like what is done in recent models [63]) reduces the maximum intermediate tensor size back to $256 \times 256 \times 8$, halving the memory footprint at each skip connection. Since both encoder and decoder features are aligned in shape, summation can be applied directly without the need for additional layers, Listing 4.3 illustrates the modified concatenation to sum PyTorch code.

```
- d5 = self.Up5(x5)
- d5 = torch.cat([x4, d5], dim=1)   # concat doubles channels
- d5 = self.Up_conv5(d5)
+ d5 = self.Up5(x5)
+ d5 = x4 + d5  # element-wise sum, shape preserved
+ d5 = self.Up_conv5(d5)

- d4 = self.Up4(d5)
- d4 = torch.cat([x3, d4], dim=1)
- d4 = self.Up_conv4(d4)
+ d4 = self.Up4(d5)
+ d4 = x3 + d4
+ d4 = self.Up_conv4(d4)

- d3 = self.Up3(d4)
- d3 = torch.cat([x2, d3], dim=1)
- d3 = self.Up_conv3(d3)
+ d3 = self.Up3(d4)
+ d3 = x2 + d3
+ d3 = self.Up_conv3(d3)

- d2 = self.Up2(d3)
- d2 = torch.cat([x1, d2], dim=1)
- d2 = self.Up_conv2(d2)
+ d2 = self.Up2(d3)
+ d2 = x1 + d2
+ d2 = self.Up_conv2(d2)
```

Listing 4.3: Decoder fusion change: concatenation to summation

Finally, at the entrance of the *Skip Fusion* block, the original grouped convolution with $g{=}2$ was replaced by a single 3D convolution. This change was made specifically because of the FPGA implementation, it avoids implementing a grouped convolution core and instead reuses a single convolution 3D core, ultimately reducing FPGA resource utilization at the cost of a small increase in parameter and MACs counts, Listing 4.4 describes the modified PyTorch convolution code in *Skip Block*.

```
 class FusionConv(nn.Module):
     def __init__(self, ch_in, ch_out, act: nn.Module = nn.Hardswish):
         super().__init__()
         self.conv = nn.Sequential(
-            nn.Conv2d(ch_in, ch_in, kernel_size=3, stride=1, padding=1, groups=2, bias=True),
+            nn.Conv2d(ch_in, ch_in, kernel_size=3, stride=1, padding=1, groups=1, bias=True),
             nn.BatchNorm2d(ch_in),
             act(),
             nn.Conv2d(ch_in, ch_out * 4, kernel_size=1),
             nn.BatchNorm2d(ch_out * 4),
             act(),
             nn.Conv2d(ch_out * 4, ch_out, kernel_size=1),
             nn.BatchNorm2d(ch_out),
             act(),
         )
```

Listing 4.4: FusionConv: remove groups=2, add 3D alternative

This modification aligns with recent trends in efficient neural network design, where balancing performance and resource usage is essential — especially for deployment in real-time [38, 46,

54, 64, 65] Figure 4.3 illustrates the final modified *Skip Fusion* block.

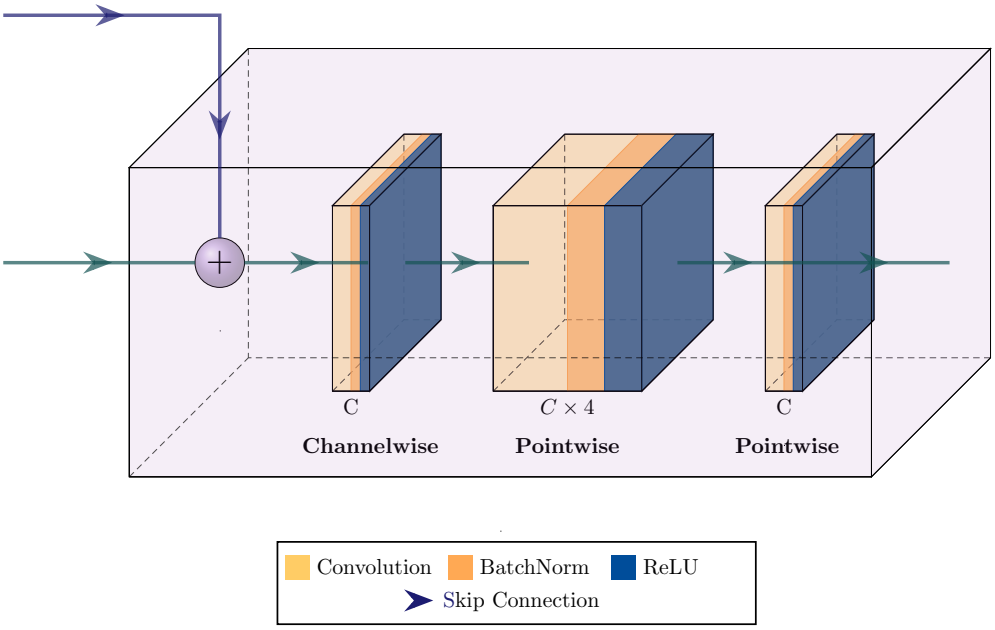


Figure 4.3: Modified *Skip Fusion* block.

4.1.6 Quantization Aware Training

To reduce computational cost and accelerate inference on FPGAs, the final model is quantized using the Brevitas framework (v0.12.0) [66]. Brevitas enables Quantization-Aware Training (QAT) by providing PyTorch-compatible modules and configurable quantizers for weights, activations, and biases. By lowering bit-widths across tensors, the model achieves a substantially smaller memory footprint and higher throughput. Further details on the quantization setup and results are presented in the next chapter (see Chapter 5).

4.2 Final Architecture

Mobile-CMUNeXt represents the culmination of architectural optimizations applied to the CMUNeXt network, designed specifically for deployment in FPGAs. Mobile-CMUNeXt balances accuracy, efficiency, and compactness. It inherits the modular structure and hierarchical design of CMUNeXt, while introducing significant reductions in parameter count and computational cost. This is achieved through a combination of channel pruning, activation function simplification, skip connection restructuring, and layer reordering, mentioned in the last chapter.

Unlike traditional segmentation networks which prioritize performance at the expense of computational cost, Mobile-CMUNeXt is engineered to retain competitive segmentation capabilities while remaining ultra-lightweight. It maintains large receptive fields via wide kernels (as used in CMUNeXt-S), but reduces block lengths and channels across the network. The result is a model with a minimal footprint suitable for real-time inference on edge devices.

Mobile-CMUNeXt follows the encoder–decoder topology just like CMUNeXt, retaining five depths (L1–L5) with symmetric skip paths. The network is parameterized by channel widths C, block lengths L, and kernel sizes K per depth. Stage widths are intentionally small and constrained to multiples of eight ($C \in \{8, 8, 16, 16, 24\}$). All modules adopt the *Conv* $\rightarrow$ *BN* $\rightarrow$ *ReLU* ordering to support batch-normalization folding.

The stem uses a 3×3 convolution (stride 1, padding 1) with batch normalization and ReLU, preserving the input resolution while projecting into a compact feature space. Downsampling is performed via 2×2 max pooling (stride 2); upsampling relies on bilinear interpolation followed by a 3×3 convolution, BN, and ReLU to refine high-resolution features. The core encoder block—*Mobile-CMUNeXt Block*—extends the original CMUNeXt design with two residual *depthwise* convolutions in sequence to enhance spatial representation with little parameter (see Figure 4.2). These are followed by a $4\times$ expansion 1×1 convolution and a projection back to the base width.

Skip connections (see Figure 4.3) are restructured to use element-wise *summation* rather than concatenation, keeping tensor shapes fixed at fusion points and halving intermediate memory compared to concatenation-based designs. Additionally, the initial grouped convolution ($g{=}2$) at the entrance of the fusion block is replaced by an early *3D convolution*: encoder and decoder features are stacked along a pseudo-depth axis ($D{=}2$) and mixed with a $k_d\times k\times k$ kernel, consolidating cross-path interaction without inflating activation sizes. ReLU is employed throughout in place of smoother activations (e.g., GELU or Hardswish), providing a favorable trade-off among accuracy, simplicity, and quantization compatibility.

The bottleneck lies between the last encoder stage and the start of the decoder — after stage $L5$ and before the first upsampling layer. At this point, four 2×2 pooling operations have reduced the feature map to its smallest size: for an input of $H\times W$, it becomes $\frac{H}{16}\times\frac{W}{16}$ (labeled $I/16$ in the diagram). In the *Mobile-CMUNeXt* setup, this stage has 24 channels ($C_5 = 24$) and keeps the same *depthwise–expansion–projection* structure as earlier encoder blocks. From here, the decoder upsamples step by step ($I/8 \rightarrow I/4 \rightarrow I/2 \rightarrow I$), using skip connections from $C4$, $C3$, $C2$, and $C1$ to recover spatial details.

The final PyTorch implementation of the Mobile-CMUNeXt network can be found at Section B.2, Figure 4.4 and Appendix E illustrate the final architecture of the network.

In total, the *Mobile-CMUNeXt* network has **55** convolutional layers, including the initial stem and final classifier. The architecture also integrates **4** max-pooling layers for downsampling and **4** upsampling layers in the decoder to restore spatial resolution. Despite its compact design, the network preserves a deep hierarchical structure capable of rich feature extraction across multiple scales.

4.3 Performance Results

After defining the model some experiments were conducted using the same testing environment described in Chapter 3. All models were trained under identical conditions with consistent preprocessing, training, and validation metrics to ensure comparability. The performance re-

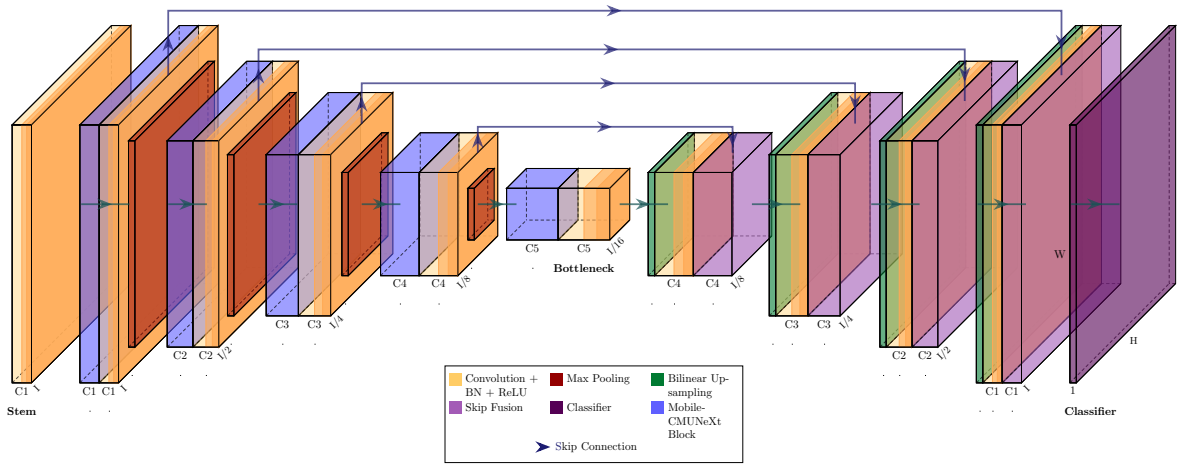


Figure 4.4: Overview of the Mobile-CMUNeXt architecture.

sults for Mobile-CMUNeXt are summarized in Table 4.6. Despite its drastically smaller size — only **0.04M parameters** and **0.49G MACs** — *Mobile-CMUNeXt* maintains highly competitive performance across all datasets.

Network	MACs (G)↓	Params (M)↓	BUSI		FIVES		ISIC2016	
			IoU↑	F1↑	IoU↑	F1↑	IoU↑	F1↑
AttentionUNet	66.63	34.88	74.53	61.9	82.08	69.75	90.76	83.4
UNet	65.52	34.53	74.89	61.1	80.18	67.0	90.59	83.0
UNet++	34.9	9.16	73.88	61.16	**89.2**	**80.54**	90.88	83.67
UNet_V2	5.1	24.9	76.37	63.28	72.94	57.43	90.94	83.73
TransUnet	32.23	105.32	77.81	65.2	87.85	78.37	91.56	84.69
CFPNetM	3.47	0.76	**79.76**	**67.1**	88.3	79.06	91.88	85.16
CMUNeXt	7.42	3.15	78.94	65.84	81.46	68.77	91.79	85.0
CMUNeXt-L	17.18	8.29	78.63	65.83	85.59	74.82	91.83	85.03
CMUNeXt-S	1.09	0.42	77.39	64.12	85.85	75.24	91.71	84.86
GIVTED-Net	0.37	0.19	77.92	64.6	86.37	76.02	**92.34**	**85.91**
LUCF-Net	8.59	6.93	79.1	66.17	76.51	62.03	92.04	85.37
ULite	0.76	0.88	78.93	66.11	77.58	63.44	91.49	84.47
UNeXt	0.57	1.47	77.56	64.01	59.72	42.71	91.32	84.18
Mobile-CMUNeXt	0.49	**0.04**	78.93	65.81	85.69	75.0	91.75	84.94

Table 4.6: Performance comparison of Mobile-CMUNeXt across medical imaging datasets.

Given the performance results the following can be deducted about Mobile-CMUNeXt:

- In **ISIC2016** dataset it achieves **84.94% F1**, nearly matching the full CMUNeXt (84.99%) and outperforming CMUNeXt-S (84.86%).
- In **BUSI** dataset it reaches **65.81% IoU**, surpassing CMUNeXt-S (64.12%) and approaching the base CMUNeXt (65.84%). Notably, it performs on par with the likes of CFPNetM, which have **over 19 times** more parameters and the best performance.
- In **FIVES** dataset it scores **75.00% IoU**, exceeding CMUNeXt (68.77%) and even sur-

passing the more heavy variant CMUNeXt-L (74.82%).

These overall results establish *Mobile-CMUNeXt* as a high-performing lightweight alternative in the domain of medical image segmentation. It demonstrates state-of-the-art performance while significantly reducing computational and memory demands.

4.4 Inference Results

Figure 4.5, Figure 4.6, and Figure 4.7 illustrate representative inference outcomes on the BUSI, FIVES, and ISIC datasets. In all overlays the *predicted segmentation* is shown in red and the *ground truth* in green, with the ground truth drawn first and the prediction on top. This rendering makes overlapping regions appear mostly red, while isolated green or red segments highlight false negatives and false positives, respectively. This visualization allows a direct qualitative assessment of how well the models align with expert annotations (ground truth mask).

Across the BUSI dataset (Figure 4.5), Mobile-CMUNeXt provides lesion outlines that follow the target fairly close. Although the raw F-Accuracy of 67.83% is lower than the best competing networks (e.g., GIVTED-Net at 85.38%), the overlay images reveal that most disagreements occur along the more ambiguous parts of the input image. Overall in this case, Mobile-CMUNeXt produces contours that are smooth and topologically correct, which is more relevant than minor pixel-level deviations.

On the ISIC dataset (Figure 4.7), where lesions are typically large, well contrasted, and have clear margins, Mobile-CMUNeXt achieves high agreement with the ground truth (F-Acc 93.46%). This performance is on par with or only slightly below the strongest baselines (e.g., TransUNet at 95.30%). The overlays demonstrate that residual errors are confined to narrow boundary bands, confirming that Mobile-CMUNeXt preserves the important shape and extent of lesions.

The FIVES dataset (Figure 4.6) is the most challenging. Veins are sparse, made up of tiny filaments, and often only a few pixels wide. Under these conditions, foreground accuracy is particularly unforgiving: even a one-pixel lateral shift can be counted as a large error. Mobile-CMUNeXt obtains an F-Acc of 56.87%, which is modest but comparable to other models. More importantly, the overlays show that Mobile-CMUNeXt recovers the main vascular branches and their connectivity, ensuring that the structural information required for clinical assessment is present. This highlights a key limitation of relying solely on F-Acc in very thin-structure segmentation tasks.

In summary, Mobile-CMUNeXt consistently delivers coherent and clinically useful segmentations. While its foreground accuracy is sometimes lower than that of larger or more complex networks, the overlays reveal that it captures the essential structures needed for aided diagnosis. Its balance of accuracy and efficiency makes it well suited for medical imaging segmentation tasks.

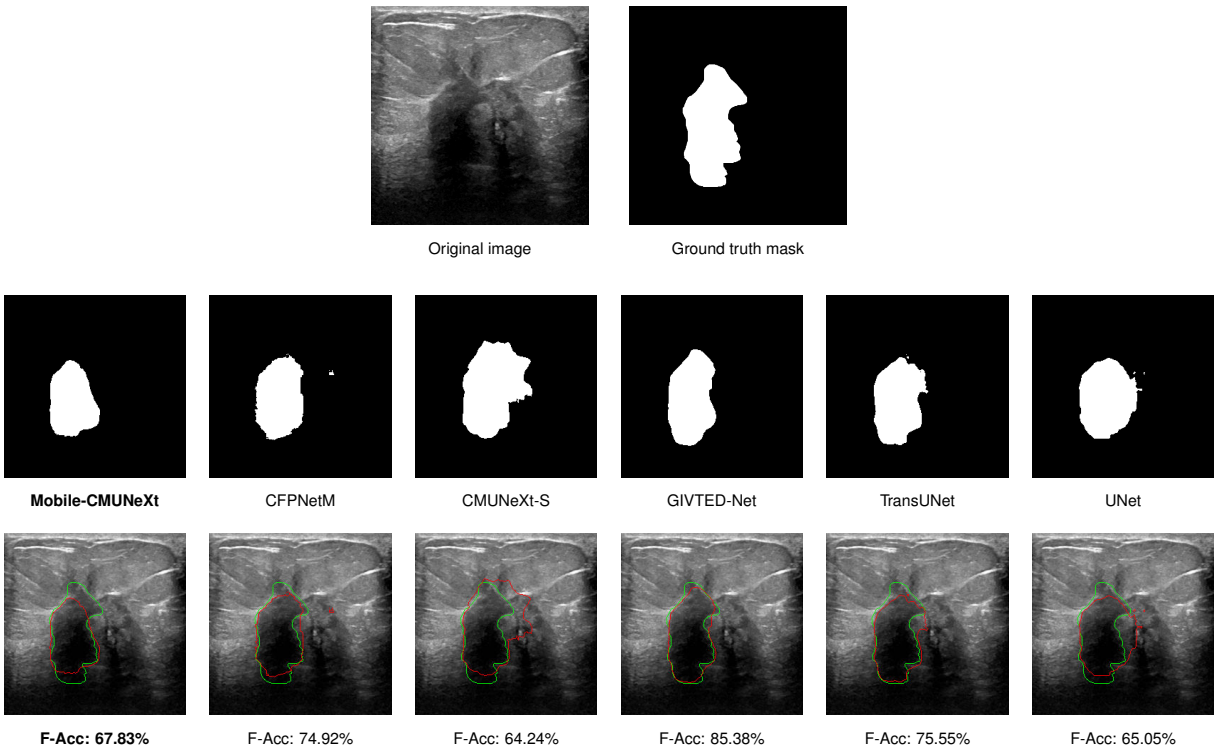


Figure 4.5: BUSI qualitative comparison. Top: original and ground truth. Middle: raw binary predictions. Bottom: overlays with color code (*prediction* red, *ground truth* green), and per-image foreground accuracy (F-Acc). Visual agreement is high along most lesion boundaries; residual errors are localized to thin, low-contrast rims.

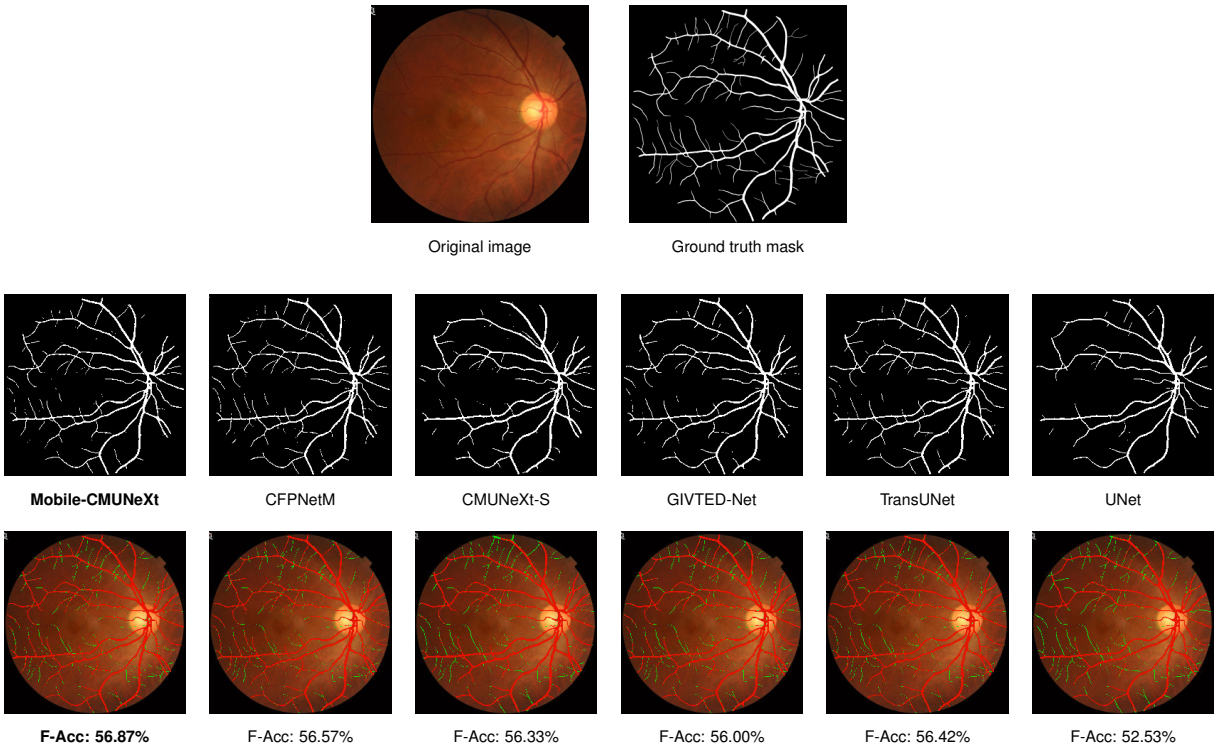


Figure 4.6: FIVES qualitative comparison. Top: original and ground truth. Middle: raw binary predictions. Bottom: overlays with color code (*prediction* red, *ground truth* green) and F-Acc. Because veins are sparse and thin small pixel shifts produce large metric swings; despite F-Acc not being close to 100%, the model captures the main vascular branches and connectivity, which is what matters for aided diagnosis.

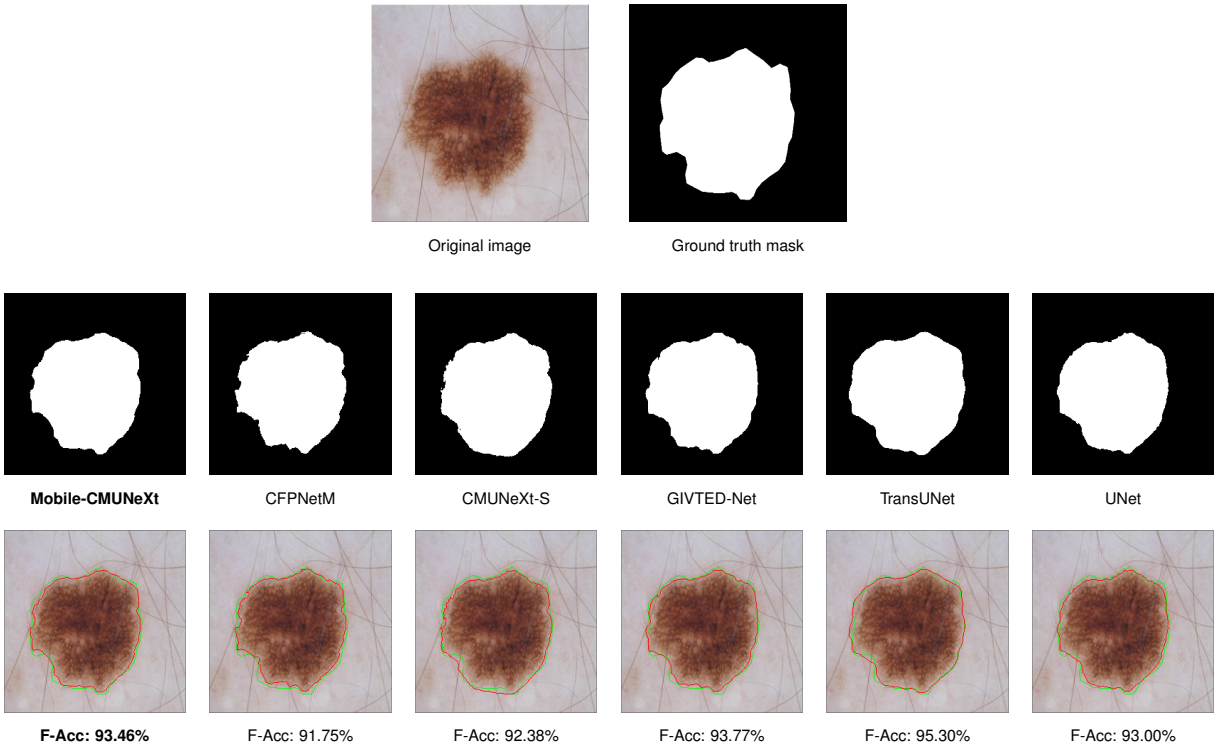


Figure 4.7: ISIC qualitative comparison. Top: original and ground truth. Middle: raw binary predictions. Bottom: overlays with color code (*prediction* red, *ground truth* green) and F-Acc. Larger, well-contrasted lesions lead to tight agreement and high F-Acc, with residual discrepancies confined to narrow boundary bands.

5 Mobile-CMUNeXt Quantization

Quantization is a process that maps floating-point value tensors to low-precision integer representations called fixed-point values in order to reduce memory footprint and arithmetic cost. This chapter delves into the quantization techniques applied to the Mobile-CMUNeXt network, including QAT, batch-normalization merge (BN merge) and weight extraction using the Brevitas (v0.12.0) framework.

The motivation behind quantization comes from the observation that deep neural networks are most of the time over-parameterized and work with higher numerical precision than necessary. By limiting weights, activations, and biases to lower bit-widths, models can achieve smaller storage requirements, faster inference, and improved deployment efficiency on resource-constrained hardware such as Field Programmable Gate Arrays (FPGAs) [67, 68].

In particular, QAT allows the model to adapt to quantization noise during training, obtaining accuracy much closer to that of the full-precision baseline compared to post-training quantization [69, 70]. Furthermore, techniques such as BN merge simplify the deployment graph by folding batch normalization layers into adjacent convolution layers, thereby eliminating runtime operations that are difficult to map efficiently to fixed-point arithmetic [71]. Finally, the extraction of quantized weights ensures that the network parameters are exported in a format directly usable by custom accelerators.

The following sections describe the quantization configuration used in this work, detail the training procedure under reduced precision, and present an evaluation of the resulting models under various bit-width settings.

5.1 Quantization-Aware Training Setup

Brevitas is a PyTorch-based library for quantization-aware training of neural networks, designed to facilitate the deployment of models on low-precision hardware. It provides a set of quantized layers, quantizers, and utilities that integrate seamlessly with PyTorch, allowing for easy experimentation with different quantization schemes. In this work Brevitas is used to implement QAT for the Mobile-CMUNeXt network, enabling the training of a quantized version of the model while maintaining high accuracy. This section details the QAT setup for transforming the Mobile-CMUNeXt network PyTorch implementation into a Brevitas compatible one. The complete quantized network definition is provided in Section C.2, and the quantization configuration (training metadata) is given in Section C.3. After this initial setup the network can be trained

using the same code used before, provided in Appendix A.

5.1.1 Bit-Width

The quantization setup begins by defining the numerical precision (bit-width) used for weights, activations, and biases. These values are set through a function `set_bit_widths(weight_bit_width, act_bit_width, bias_bit_width)` (Section C.3) and stored as global parameters applied across the network. By default, the configuration uses 8-bit weights and activations and 16-bit biases, though any precision can be specified to explore the effect of reduced bit-width (Section 5.6.1). The helper functions `get_weight_bit_width()` and `get_act_bit_width()` bind each quantized layer to the active precision setting (Listing 5.1).

```
# quant_config.py
def set_bit_widths(weight_bit_width=8, act_bit_width=8, bias_bit_width=16):
    global WEIGHT_BIT_WIDTH, ACT_BIT_WIDTH, BIAS_BIT_WIDTH
    WEIGHT_BIT_WIDTH = weight_bit_width
    ACT_BIT_WIDTH    = act_bit_width
    BIAS_BIT_WIDTH   = bias_bit_width

def get_weight_bit_width(): return WEIGHT_BIT_WIDTH
def get_act_bit_width():    return ACT_BIT_WIDTH
```

Listing 5.1: Runtime selection of weight/activation/bias bit-widths

5.1.2 Quantizer Definitions

The next step is to define the Brevitas quantizers used in each layer. Brevitas attaches *quantizers* to parameters and activations, enforcing low-precision representations during training while retaining floating-point copies for optimization. This mechanism simulates fixed-point behavior and propagates quantization metadata across layers.

Quantizers are defined using `ExtendedInjector`-based classes. The base policy `BaseQuant` applies: constant bit-widths (`BitWidthImplType.CONST`), stats-based scaling using the tensor's maximum value (`ScalingImplType.STATS + StatsOp.MAX`), power-of-two scale restriction (`RestrictValueType.POWER_OF_TWO`) for FPGA-friendly shifts, zero offset (`ZeroZeroPoint`), round-to-nearest conversion (`FloatToIntImplType.ROUND`), and per-tensor scaling. Derived classes `WeightQuant`, `ActQuant`, and `BiasQuant` apply these settings consistently across kernels, activations, and biases respectively (Section C.3). A summarized explanation of each policy can be found in Table 5.1.

5.1.3 Layer Replacement

Finally, to enable QAT, a quantized variant of the Mobile-CMUNeXt PyTorch model is defined in which floating-point layers are swapped for Brevitas quantized counterparts.

A `qnn.QuantIdentity` node is inserted at the model input and immediately before each residual/skip summation so that *QuantTensor* metadata is propagated and operands are aligned prior to addition; without these identities, element-wise sums would suffer from inconsistent scaling.

Aspect	Setting	Notes / Rationale
Bit-width policy	`BitWidthImplType.CONST`	Constant bit-widths during training (typ. W=8, A=8, B=16).
Scaling method	`ScalingImplType.STATS` + `StatsOp.MAX`	Scale s computed from (abs) max of tensor; no learned scale. Utilizes full dynamic range.
Scale restriction	`RestrictValueType.POWER_OF_TWO`	Enables shift-based requantization on FPGAs; cheap hardware divides/multiplies.
Zero-point	`ZeroZeroPoint`	Symmetric affine quantization with $z = 0$.
Float→int cast	`FloatToIntImplType.ROUND`	Round-to-nearest before clipping; stable during QAT.
Granularity	`scaling_per_output_channel=False` (default)	Per-tensor scaling; optionally use per-output-channel for weights when required.
Quantizer classes	`WeightQuant`, `ActQuant`, `BiasQuant`	Apply policies to kernels, activations, and biases; align bit-width/range across the network.

Table 5.1: Brevitas quantizer configuration used in Mobile-CMUNeXt QAT.

The complete mapping between standard `torch.nn` operators and their quantized equivalents is shown in Table 5.2, with the full implementation provided in Section C.2.

Original (torch.nn)	Brevitas (brevitas.nn)
`nn.Conv2d`	`qnn.QuantConv2d`
`nn.Upsample(scale_factor=2, mode=up_mode)`	`qnn.QuantUpsamplingBilinear2d`
`nn.ReLU`	`qnn.QuantReLU`
`nn.Identity`	`qnn.QuantIdentity`

Table 5.2: Layer mapping from floating-point `torch.nn` to Brevitas `brevitas.nn` used for QAT.

5.2 Batch Normalization Folding

Given that layers now follow the *Conv* $\rightarrow$ *BN* $\rightarrow$ *Activation* order (see Section 4.1.3), the batch normalization (BN) operation can be absorbed into the preceding convolution at export time. This optimization removes the BN layer, reducing both computational cost and memory access.

A batch normalization layer is defined by learnable parameters γ (scale) and β (shift), as well as the running mean μ and variance σ^2, computed during training. The small constant ϵ ensures numerical stability. Given a convolution with weights W and bias b, followed by BN, the normalized output is:

$$y = \gamma \cdot \frac{(Wx + b - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta.$$

To fold the BN into the convolution, the parameters are merged as:

$$W' = W \cdot \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}, \qquad b' = (b - \mu) \cdot \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} + \beta. \tag{5.1}$$

Here, W' and b' represent the updated convolution weights and biases after folding. Once these new parameters are applied, the BN layer becomes redundant and can be safely removed from

the model.

The practical implementation traverses the module hierarchy, identifies every `Conv2d/QuantConv2d` followed by a `BatchNorm2d`, merges their parameters, and replaces the BN layer with an identity operation. The folding procedure is illustrated in Listing 5.2.

```
1  # 'butils.merge_bn' applies standard BN folding to QuantConv2d + BatchNorm2d.
2  def merge_bn_with_brevitas(conv, bn, conv_name, bn_name):
3      print(f"Merging BN '{bn_name}' into Conv '{conv_name}'")
4      butils.merge_bn(conv, bn)
5      # Reset BN to identity so it has no residual effect
6      bn.reset_parameters()
7      with torch.no_grad():
8          bn.weight.fill_(1.0)
9          bn.bias.zero_()
10         bn.running_mean.zero_()
11         bn.running_var.fill_(1.0)
12
13 def fold_bn_recursively(model):
14     prev_module = None
15     prev_name = None
16     for name, module in model.named_children():
17         # Recurse through submodules
18         fold_bn_recursively(module)
19
20         # Detect Conv/BN pattern
21         if isinstance(prev_module, (nn.Conv2d, qnn.QuantConv2d)) \
22            and isinstance(module, nn.BatchNorm2d):
23             merge_bn_with_brevitas(prev_module, module, prev_name, name)
24             setattr(model, name, nn.Identity())
25         prev_module, prev_name = module, name
26     return model
```

Listing 5.2: Recursive BN folding for Conv2d/QuantConv2d → BatchNorm2d pairs.

At line 22, the function identifies a convolution immediately followed by a BN layer. The actual merging is performed in line 23, which calls the Brevitas utility (line 4). Finally, the BN is replaced by an identity operation at line 24, ensuring the model graph remains consistent. To prevent any residual effect from remaining BN parameters, they are reset to identity values between line 6–line 9.

After folding, a quantized Mobile-CMUNeXt variant is generated in which all `BatchNorm2d` layers are replaced with `QuantIdentity(act_quant=None, return_quant_tensor=True)` modules. This represents the deployment-ready architecture: batch normalization layers are absorbed into their preceding convolutions, simplifying computation and reducing inference latency. The `QuantIdentity` layers preserve quantization metadata to maintain correct scaling across the network.

Finally, a brief fine-tuning stage (50 epochs) is performed using the pretrained weights to compensate for small numerical shifts introduced by folding and quantization noise. This allows the quantized model to recover its baseline accuracy and, in some cases, even slightly outperform the original floating-point version.

5.3 Exporting Quantized Parameters

Exporting quantized weights and biases is a crucial step for deploying the trained Mobile-CMUNeXt model on a FPGAs. The goal is to convert and export the learned parameters into fixed-point integers. Brevitas stores floating-point parameters together with quantization metadata (bit-width, scale, sign). The exporter (See Section C.4) performs fixed-point casting, layer-type–specific tensor reordering, and emission of C headers for HLS build-time inclusion.

5.3.1 Fixed-point casting utilities

Listing 5.3 shows the core helpers: `float_to_fixed_array` multiplies by $2^{\text{frac\_bits}}$ and rounds to the target integer type (in this works context 8-bits for weights and 16-bits for bias), while `bit_width_to_dtype` maps the Brevitas bit-width to a NumPy integer dtype.

```python
def float_to_fixed_array(values, scale, dtype=np.int8):
    values = np.asarray(values, dtype=np.float32)
    scale = np.asarray(scale, dtype=np.float32)
    q = np.round(values / scale)
    return q.astype(dtype)

def bit_width_to_dtype(bit_width, signed=True):
    dtype_map = {
        8:  (np.int8,  np.uint8),
        16: (np.int16, np.uint16),
        32: (np.int32, np.uint32),
        64: (np.int64, np.uint64),
    }
    if bit_width not in dtype_map:
        raise ValueError(f"Unsupported bit width: {bit_width}.")
    return dtype_map[bit_width][0] if signed else dtype_map[bit_width][1]
```

Listing 5.3: Fixed-point casting utilities.

5.3.2 FPGA-Friendly Tensor Layouts

To align with each hardware core's memory access pattern, convolutional weights must be re-ordered before export (see the following chapter for hardware details). Listing 5.4 defines three specializations of the `ParametersCppBase` class, each corresponding to a specific convolution type and layout transformation.

- **Pointwise (PW):** squeezes singleton spatial dimensions, reshaping $(\text{outC}, \text{inC}, 1, 1) \rightarrow (\text{outC}, \text{inC})$.
- **Depthwise (DW):** moves the channel dimension to the last axis, $(\text{outC}, 1, k, k) \rightarrow (k, k, \text{outC})$, to match per-channel parallelism in hardware.
- **3D (standard convolution):** interleaves output and input channels, $(\text{outC}, \text{inC}, k, k) \rightarrow (k, k, \text{outC}, \text{inC})$, optimizing data locality for 3D compute cores.

The abstract base class handles the emission of C-style array initializers and tracks tensor dimensions to automatically generate headers with the proper array declarations for each layer.

```python
class ParametersPWCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weightsPW.h", "biasPW.h")
    def handle_weights(self, fixed_weights):
        return np.squeeze(fixed_weights, axis=(2, 3))  # (outC,inC,1,1)->(outC,inC)

class ParametersDWCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weightsDW.h", "biasDW.h")
    def handle_weights(self, fixed_weights):
        return np.squeeze(fixed_weights, axis=1).transpose(1, 2, 0)  # ->(kH,kW,outC)

class Parameters3DCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weights3D.h", "bias3D.h")
    def handle_weights(self, fixed_weights):
        return fixed_weights.transpose(2, 3, 0, 1)  # ->(kH,kW,outC,inC)
```

Listing 5.4: Writers that reshape tensors and emit C arrays for each convolution type.

Each class specialization ensures that exported tensors conform to the expected hardware layout for its corresponding convolution engine. This step is crucial for efficient streaming and memory alignment during FPGA execution, minimizing data rearrangement overhead at runtime.

5.3.3 Traversing the Model and Exporting

The main export routine (Listing 5.5) iterates through all modules in the network, identifies quantized convolution layers, extracts their *quantized* parameters from Brevitas' `QuantTensor`, converts them to fixed-point format, and writes both binary data and C headers for hardware integration.

For each module, the exporter determines the appropriate writer class — depthwise (DW), pointwise (PW), or 3D convolution — based on the kernel shape and grouping configuration. It then reads quantization metadata (scale, sign, and bit-width), converts tensors to integers using `float_to_fixed_array` (Listing 5.3), and stores the results in multiple formats: binary files, debug logs, and C headers. The C headers (`weightsDW.h`, `biasDW.h`, etc.) are automatically closed by the `cleanup()` function once the traversal completes.

```python
def extract_weights_and_bias(model, output_dir, device):
    os.makedirs(output_dir, exist_ok=True)
    id = 0

    weightsDWCpp = ParametersDWCpp(output_dir)
    weightsPWCpp = ParametersPWCpp(output_dir)
    weights3DCpp = Parameters3DCpp(output_dir)
    weights_cpp_class : ParametersCppBase = None

    for name, module in model.named_modules():
        base_name = f"{id}_{name}_{module.__class__.__name__}"
        description_file_name = f"{base_name}.txt"
        debug_file_name =  f"{base_name}.values"
        bin_file_name = f"{base_name}.bin"

        delete_if_found(description_file_name)
```

```
17          delete_if_found(bin_file_name)
18          delete_if_found(debug_file_name)
19
20          # --- Select writer by layer type ---
21          if isinstance(module, QuantConv2d):
22              if module.groups == module.in_channels and module.in_channels == module.
                    out_channels:
23                  weights_cpp_class = weightsDWCpp
24              elif module.kernel_size == (1, 1) and module.groups == 1:
25                  weights_cpp_class = weightsPWCpp
26              else:
27                  weights_cpp_class = weights3DCpp
28              # === Weights ===
29              if hasattr(module, "weight") and hasattr(module, "weight_quant"):
30                  id+=1
31                  quant_tensor = module.quant_weight()
32                  scale = getattr(quant_tensor, "scale", None)
33                  scale = scale.item() if hasattr(scale, 'item') else scale
34                  signed = getattr(quant_tensor, "signed_t", None)
35                  weights = getattr(quant_tensor, "value", None)
36                  bit_width = getattr(quant_tensor, "bit_width", None)
37                  dtype = bit_width_to_dtype(int(bit_width), signed)
38                  weights_np = weights.detach().to(device).numpy()
39                  fixed_values = float_to_fixed_array(weights_np, scale, dtype)
40                  # Write debug and binary files
41                  with open(bin_file_name, "ab") as f:
42                      f.write(fixed_values.tobytes())
43                  # Write to C header
44                  weights_cpp_class.write_weights(fixed_values)
45              # === Bias ===
46              if hasattr(module, "bias_quant"):
47                  quant_tensor = module.quant_bias()
48                  scale = getattr(quant_tensor, "scale", None)
49                  scale = scale.item() if hasattr(scale, 'item') else scale
50                  signed = getattr(quant_tensor, "signed_t", None)
51                  bias = getattr(quant_tensor, "value", None)
52                  bit_width = getattr(quant_tensor, "bit_width", None)
53
54                  bias_np = bias.detach().to(device).numpy()
55                  array_flat = bias_np.flatten(order="C")
56
57                  dtype = bit_width_to_dtype(int(bit_width), signed)
58                  fixed_values = float_to_fixed_array(array_flat, scale, dtype)
59
60                  with open(bin_file_name, "ab") as f:
61                      f.write(fixed_values.tobytes())
62                  weights_cpp_class.write_bias(fixed_values)
63
64      weightsDWCpp.cleanup()
65      weightsPWCpp.cleanup()
66      weights3DCpp.cleanup()
```

Listing 5.5: Export loop: traverse model, extract quantized tensors, and emit C headers.

At line 10, the function traverses the full model hierarchy using `model.named_modules()`. The writer type is selected at line 27 depending on whether the layer is depthwise, pointwise, or 3D. Quantization metadata is retrieved from Brevitas' `QuantTensor` (line 32–line 36), and tensor values are converted from floating-point to fixed-point integers at line 39. Binary arrays are written at line 42, while the converted parameters are appended to their corresponding C headers in line 44. Bias values follow the same process (line 61), and finally, the header files are properly

terminated at line 66 via the `cleanup()` function.

In summary, the exporter walks through the quantized model, extracts all quantization metadata (scale, bit-width, and sign), performs fixed-point conversion, and writes the results in both debug and C-compatible formats. The output includes `weightsDW.h/biasDW.h`, `weightsPW.h/biasPW.h`, and `weights3D.h/bias3D.h`, which are directly included in the FPGA build for deployment.

5.4 Exporting Shift Scales

Brevitas produces per-tensor *scales* as part of its quantization metadata, mapping integer activations to real-valued ranges learned during QAT. On hardware, these scale transitions are efficiently implemented as *power-of-two* right shifts (acc $\gg s$), avoiding multipliers or dividers at runtime. Each convolution layer accumulates results in an extended integer precision, so its accumulator scale (s_{conv}) usually differs from the scale expected by the next consumer (e.g., `QuantReLU` or final `QuantIdentity`). The required integer shift between them is computed as

$$s = \text{round}\left(\log_2 \frac{s_{\text{out}}}{s_{\text{conv}}}\right), \tag{5.2}$$

where s_{conv} is the convolution output scale and s_{out} the consumer scale. This value is exported as a signed integer and later applied as a right bit-shift (acc $\gg s$) before clamping or activation on the FPGA. Computing s relative to the consumer ensures that each hardware core outputs activations aligned to the quantization range of the following layer.

Listing 5.6 defines the C-header emitters responsible for writing these integer shifts. Each specialization corresponds to a convolution type (pointwise, depthwise, or 3D). The base class handles file creation, appending new shift entries as they are discovered, and finalizing the arrays when all shifts have been emitted. At line 9, the current shift is appended to the array, while line 20 terminates the header file with closing braces and a total count.

```
1  class ScaleShiftCppBase:
2      def __init__(self, output_dir, scale_filename):
3          self.scale_name = os.path.splitext(scale_filename)[0]
4          os.makedirs(output_dir, exist_ok=True)
5          self.scale_file = os.path.join(output_dir, scale_filename)
6          if os.path.exists(self.scale_file): os.remove(self.scale_file)
7          self.s_counter, self._total_vals = 0, 0
8
9      def write_scale_shift(self, shift):
10         arr = np.atleast_1d(np.asarray(shift, dtype=np.int32))
11         with open(self.scale_file, "a") as f:
12             if self.s_counter == 0:
13                 f.write("// Auto-generated from quantized model\n")
14                 f.write(f"static const int {self.scale_name}[] = {{")
15             f.write(f"\n/* idx: {self.s_counter}, count: {arr.size} */\n    ")
16             f.write(", ".join(map(str, arr)) + ",")
17         self.s_counter += 1
18         self._total_vals += arr.size
19
20     def cleanup(self):
21         if self.s_counter > 0:
22             with open(self.scale_file, "a") as f:
```

```
23                 f.write("\n};\n")
24                 f.write(f"// total values: {self._total_vals}\n")
25
26 class ScaleShiftPWCpp(ScaleShiftCppBase):  # scalePW.h
27     def __init__(self, output_dir): super().__init__(output_dir, "scalePW.h")
28 class ScaleShiftDWCpp(ScaleShiftCppBase):  # scaleDW.h
29     def __init__(self, output_dir): super().__init__(output_dir, "scaleDW.h")
30 class ScaleShift3DCpp(ScaleShiftCppBase):  # scale3D.h
31     def __init__(self, output_dir): super().__init__(output_dir, "scale3D.h")
```

Listing 5.6: Emitters that generate C header files for integer scale shifts.

To extract and compute these shift values, forward hooks are registered on every `QuantConv2d`, `QuantReLU`, and the final `QuantIdentity`. Each convolution hook records its output scale and convolution type (DW, PW, or 3D), while the activation hook retrieves the consumer's scale and computes the integer shift using Equation 5.2. The relevant code is shown in Listing 5.7. At line 15, the convolution hook captures the accumulator scale; the shift computation occurs at line 22; and the activation hooks (line 27–line 31) write the resulting shift to the proper emitter. Any convolution without a matching consumer emits a zero shift during cleanup (line 42).

```
1  def _scale_of(obj):
2      s = getattr(obj, "scale", None)
3      if s is None: return None
4      return float(s.item()) if hasattr(s, "item") else float(s)
5
6  def attach_hooks(model: nn.Module, output_dir="./output"):
7      scaleDWCpp, scalePWCpp, scale3DCpp = (
8          ScaleShiftDWCpp(output_dir),
9          ScaleShiftPWCpp(output_dir),
10         ScaleShift3DCpp(output_dir),
11     )
12     name_by_module = {m: n for n, m in model.named_modules()}
13     conv_info_stack = deque()
14
15     def conv_hook(module, inputs, output):
16         if module.groups == module.in_channels: kind = "DW"
17         elif module.kernel_size == (1,1) and module.groups == 1: kind = "PW"
18         else: kind = "C3D"
19         s_conv = _scale_of(output)
20         conv_info_stack.append({"kind": kind, "s_conv": s_conv})
21
22     def _write_shift_from(conv_info, s_out):
23         s_in, kind = conv_info["s_conv"], conv_info["kind"]
24         shift = 0 if not s_in or not s_out or s_in<=0 or s_out<=0 else int(round(math.log2(
               s_out/s_in)))
25         {"DW": scaleDWCpp, "PW": scalePWCpp, "C3D": scale3DCpp}[kind].write_scale_shift(shift
               )
26
27     def relu_hook(module, inputs, output):
28         if conv_info_stack:
29             _write_shift_from(conv_info_stack.pop(), _scale_of(output))
30
31     def qid_hook(module, inputs, output):
32         name = name_by_module.get(module, "")
33         if name == "output_quant" or name.endswith(".output_quant"):
34             if conv_info_stack:
35                 _write_shift_from(conv_info_stack.pop(), _scale_of(output))
36
37     for m in model.modules():
38         if isinstance(m, qnn.QuantConv2d):   m.register_forward_hook(conv_hook)
```

```
        elif isinstance(m, qnn.QuantReLU):   m.register_forward_hook(relu_hook)
        elif isinstance(m, qnn.QuantIdentity): m.register_forward_hook(qid_hook)

    def cleanup():
        while conv_info_stack:
            _write_shift_from(conv_info_stack.popleft(), s_out=None)
        for w in (scaleDWCpp, scalePWCpp, scale3DCpp): w.cleanup()
    return cleanup
```

Listing 5.7: Hooks that pair convolution and consumer scales and export shift integers.

The main execution script (Listing 5.8) loads the quantized model, performs a warm-up forward pass to ensure all quantized tensors carry valid scale values, attaches hooks, and executes another forward pass to emit the shift headers. Finally, the `cleanup()` call finalizes all generated C arrays.

```
def conv_scale_extractor(device, model_dir):
    config = load_yaml(os.path.join(model_dir, "config.yml"))
    model_pth = os.path.join(model_dir, "model.pth")
    model = load_model(config, model_pth, device)
    input_ = generate_input(config, device)
    output_dir = os.path.join(model_dir, "weights")

    model(input_)                       # warm-up: populate cached scales
    cleanup = attach_hooks(model, output_dir)
    model(input_)                       # hooked pass: emit shifts
    cleanup()                           # finalize header files
```

Listing 5.8: Load quantized model, collect scales, and export integer shift headers.

On the FPGA, each convolution core performs its multiply–accumulate operations in widened precision, then applies the exported integer shift before activation. This ensures that every core's output is correctly aligned to its consumer's quantization scale, enabling seamless chaining of quantized layers without any runtime scale computation overhead.

5.5 Describing Quantized Model

After QAT, it is useful to generate a description of the quantized model in order to verify how quantization was applied across layers. This description provides a layer-by-layer account of the network, including shapes, quantization parameters, and derived fixed-point formats. This information is critical for both debugging and hardware deployment, as it contains the metadata necessary for effective deployment, most notably the tensor scale at each layer.

To automate this process, a Python script was developed (see Section C.6). The script loads the trained quantized model from a given directory and attaches *forward hooks* to each quantized layer. These hooks will later interpret the tensor values in each layer. The script runs a dummy forward pass and intercepts inputs, weights, bias and outputs tensor and logs their shapes and quantization metadata. After that the script converts the metadata into fixed-point formats ($Q_{m.n}$), where m and n represent the integer and fractional bit allocations, respectively. Finally, it Writes the full layer-by-layer description to a text file, which serves as a compact reference for inspection and hardware mapping. An example of the output of the file is shown below:

```
Module: input_quant (QuantIdentity)
  * Input shape: torch.Size([1, 3, 256, 256])
  * Quant Output shape: (1, 3, 256, 256), bit_width=8.0, fixed-point format=Q0.7, scale
      =6.103515625e-05, signed=True

Module: stem.conv.0 (QuantConv2d)
  * Quant Input shape: (1, 3, 256, 256), bit_width=8.0, fixed-point format=Q0.7, scale
      =6.103515625e-05, signed=True
  * Quant Weight shape: (8, 3, 3, 3), bit_width=8.0, fixed-point format=Q7.0, scale=1.0,
      signed=True
  * Quant Bias shape: (8,), bit_width=16.0, fixed-point format=Q1.14, scale=6.103515625e-05,
      signed=True
  * Quant Output shape: (1, 8, 256, 256), bit_width=22.0, fixed-point format=Q7.14, scale
      =6.103515625e-05, signed=True
```

This file documents the bit-width, scale, and fixed-point format of each layer, helping connect the high-level PyTorch training model with its hardware implementation. The file acts as both a verification tool — ensuring that quantization was correctly applied — and a design reference that guides how tensors and parameters should be represented in the accelerator cores. In practice, this means the quantized model description is not only useful for debugging and validation, but also serves as a blueprint for aligning software training outputs with the numerical precision constraints of the hardware.

5.6 Quantization Results

This section presents the evaluation of Mobile-CMUNeXt under different quantization strategies. The goal is to assess how reducing numerical precision impacts segmentation accuracy across the datasets, and to determine the most suitable configuration for FPGA deployment.

An ablation study on bit-widths is conducted, analyzing trade-offs between efficiency and accuracy. Subsequently, the effect of batch-normalization merging and fine-tuning is evaluated, comparing the quantized models against the non-quantized baseline to confirm whether the performance remains competitive after quantization-aware training or not.

5.6.1 Bit-width Experiments

To evaluate the effect of numerical precision on segmentation performance, multiple quantization settings were tested by varying the bit-widths of weights, biases, and activations. The results across the datasets are reported in Table 5.3.

The result reveal that reducing the weight bit-width from 8 to 4 maintains competitive accuracy, with a small drop in IoU and F1 across all datasets. However, pushing further down to 2 bits results in a more significant performance loss, especially on BUSI and FIVES. Lowering the bias precision to 8 bits introduces severe degradation, highlighting the importance of keeping sufficient bias precision to preserve representational capacity. Also, by reducing activations to 4 bits effectively collapses the network.

Considering these results, the configuration with **8-bit weights, 16-bit biases, and 8-bit activations (8,16,8)** provides the best balance between efficiency and accuracy. This setup

Quantization (Bits)			BUSI		FIVES		ISIC2016	
Weight	Bias	Activation	F1↑	IoU↑	F1↑	IoU↑	F1↑	IoU↑
8	16	8	76.49	**65.21**	**86.88**	**76.91**	**91.0**	**84.01**
4	16	8	**76.55**	65.01	84.8	73.76	90.85	83.86
2	16	8	75.31	63.73	81.38	68.85	90.45	83.18
8	16	**4**	0.27	0.11	0.14	0.04	0.30	0.18
8	**8**	8	12.34	7.83	30.45	22.66	87.12	83.18

Table 5.3: Comparison of Mobile-CMUNeXt segmentation performance under different quantization configurations. The last row reports the baseline results of the full-precision (non-quantized) model.

achieves the highest or second-highest scores across the experiments (as it should), without introducing much performance degradation compared to the non quantized model. Therefore, this configuration was chosen as the precision setting for subsequent experiments and for FPGA deployment.

5.6.2 Fine-tune Results

After identifying the (8,16,8) configuration as the most effective precision setting, additional experiments were conducted to study the effect of batch-normalization merging and fine-tuning. The results are summarized in Table 5.4, where each row corresponds to a different combination of applied steps (Quantization, BN-Merge, Fine-tune).

Mobile-CMUNeXt			BUSI		FIVES		ISIC2016	
Quant	BN-Merge	Fine-tune	F1↑	IoU↑	F1↑	IoU↑	F1↑	IoU↑
X	—	—	76.49	65.21	86.88	76.91	91.00	84.01
X	**X**	—	65.39	45.22	70.12	61.51	83.03	71.54
X	**X**	**X**	77.82	64.42	**87.65**	**78.04**	91.71	84.89
—	—	—	**78.93**	**65.81**	85.69	75.0	**91.75**	**84.94**

Table 5.4: Segmentation performance of Mobile-CMUNeXt under different training and quantization conditions. Columns indicate whether quantization (Quant), batch-normalization merging (BN-Merge), and fine-tuning (Fine-tune) were applied. The last row corresponds to the non-quantized floating-point baseline.

The results indicate that quantization alone already produces competitive performance compared to the full-precision baseline, with only a small loss on BUSI. On FIVES and ISIC2016, quantization preserves or even slightly improves performance relative to the baseline.

When batch-normalization merging and fine-tuning are introduced, the model maintains this high level of accuracy. On BUSI, the fine-tuned version recovers part of the gap to the baseline (F1: 77.82% vs. 78.93%). On FIVES, the fine-tuned quantized model improves over the baseline in both F1 (87.65% vs. 85.69%) and IoU (78.04% vs. 75.0%). Finally, on ISIC2016, the results remain nearly indistinguishable from the floating-point reference (F1: 91.71% vs.

91.75%).

Overall, these experiments show that the combination of quantization, BN merging, and fine-tuning represents a model that is practically equivalent to the non-quantized version across all datasets, while allowing for a hardware-friendly fixed-point representation. This confirms that the chosen configuration (8,16,8) together with a brief fine-tuning stage is a suitable trade-off between efficiency and accuracy for FPGA deployment.

5.7 Image Quantization Preprocessing

Before inference on the FPGA cores, input images must be resized, normalized, quantized to 8-bit codes, and exported in a layout compatible with the hardware input path. The preprocessing pipeline proceeds as follows: (i) resize and color-space conversion; (ii) normalization to a unit range; (iii) per-tensor input quantization using the same Brevitas policy as during QAT; and (iv) export of the resulting 8-bit binary buffers.

Each image is read and converted to RGB, then resized to the network's configured resolution and normalized. The resulting tensor has shape $[1, H, W, C]$ with `float32` values in a unit range suitable for the input quantizer, as shown in Listing 5.9. The resize and normalization occur at line 3–line 4, RGB conversion at line 7, and tensor creation at line 8–line 10.

```
1  def preprocess_image(config, image_path):
2      transform = Compose([
3          Resize(config["input_h"], config["input_w"]),
4          Normalize(),                                    # default: scale to [0,1]
5      ])
6      img = cv2.imread(image_path)
7      img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
8      img = transform(image=img)["image"]                  # H,W,C
9      img = img.astype(np.float32) / 255.0
10     img = torch.tensor(img, dtype=torch.float32).unsqueeze(0)    # [1,H,W,C]
11     return img
```

Listing 5.9: Image preprocessing: resize, normalize, RGB, tensorize.

The input quantizer mirrors the training configuration: 8-bit, symmetric ($z{=}0$), per-tensor scale restricted to powers of two, with rounding-to-nearest and statistics-based scaling from the tensor's maximum. After preprocessing, the tensor is passed through `InputQuant` to obtain a Brevitas `QuantTensor`. Values are converted to signed `int8` by dividing by the per-tensor scale and clamping to the narrow range $[-127, 127]$, as shown in Listing 5.10. The quantized value and scale are accessed at line 2, clamping limits are defined at line 3, and rounding, division, and casting occur at line 4.

```
1  def quanttensor_to_int8_bytes(qt):
2      v = qt.value.detach(); s = qt.scale.detach()
3      qmin, qmax = -127, 127
4      q = torch.round(v / s).clamp_(qmin, qmax).to(torch.int8)
5      return q.cpu().numpy().flatten(order="C").tobytes()
```

Listing 5.10: Convert a Brevitas `QuantTensor` to signed int8 bytes.

For each image, only the quantized `int8` buffer is exported. The driver in Listing 5.11 loads the trained input quantizer weights from the model checkpoint (filtering `input_quant.*` keys at line 3), runs preprocessing (line 7), applies input quantization (line 9), and writes the `int8` binary buffer using Listing 5.10 (line 12).

```
1  def export_inputs(model_dir, images, device):
2      state_dict = torch.load(os.path.join(model_dir, "model.pth"), map_location=device,
           weights_only=True)
3      filtered = {k: v for k, v in state_dict.items() if k.startswith("input_quant.")}
4      quant = InputQuant().to(device); quant.load_state_dict(state_dict=filtered, strict=True);
            quant.eval()
5
6      for img_path in images:
7          img = preprocess_image(load_yaml(os.path.join(model_dir,"config.yml")), str(img_path)
               )
8          with torch.no_grad():
9              qt = quant(img.to(device))
10         quant_out = img_path.with_name(img_path.stem + "_quant_int8.bin")
11         with open(quant_out, "wb") as f:
12             f.write(quanttensor_to_int8_bytes(qt))
```

Listing 5.11: Export 8-bit quantized input buffers aligned with the trained input quantizer.

In summary, images are resized and normalized, then quantized with the trained input quantizer so that exported `int8` codes match the QAT configuration (power-of-two scale, symmetric zero-point, narrow range). The resulting binary files can be directly loaded by the FPGA, ensuring consistency between the software preprocessing and the hardware quantization pipeline.

6 Hardware Acceleration

This chapter presents the end-to-end flow for accelerating Mobile-CMUNeXt on Xilinx devices and discusses the hardware architecture choices that make the network deployment possible on a FPGA. The design emphasizes a small set of reusable compute cores developed with AMD/Xilinx tools.

Deploying deep neural networks on Programmable Logic (PL) devices requires balance between compute density, memory bandwidth, and numerical precision [72]. Unlike GPUs, which exploit large-scale parallelism and high-bandwidth memory, FPGAs provide fine-grained control over data paths and on-chip memory. This enables implementations that are both energy-efficient and latency-aware, but also introduces challenges such as limited resources, and the need to manage dataflow explicitly.

As outlined in Chapter 4, optimizations were applied to adapt Mobile-CMUNeXt for hardware execution under these constraints. Furthermore, Chapter 5 described the quantization strategies that reduce arithmetic cost and align the network with fixed-point inference. Building on these foundations, this chapter shifts the focus to hardware, describing custom cores and their integration, and the end-to-end deployment flow that enables the quantized model to run on a FPGA. To meet the resource and performance requirements, the accelerator is composed of modular hardware blocks. These blocks form the backbone of the architecture and can be reconfigured and reused across different layers of the network.

The sections that follow detail the design of these core blocks, their dataflow optimizations, and the integration strategy used to deploy them into a complete inference pipeline.

6.1 Toolchain and Workflow

The implementation follows the standard Xilinx flow, which combines high-level synthesis, system integration, and software deployment across the Vitis and Vivado toolchain. The process proceeds through four main stages:

1. **Vitis HLS:** Accelerator kernels are described in C/C++ and verified using a testbench. This stage includes functional simulation, High-Level Synthesis (HLS) synthesis, C/Register-Transfer Level (RTL) co-simulation, and packaging of the kernels as reusable IP cores.

2. **Vivado:** The IP cores are integrated into a block design together with on-chip memory,

interconnects, and Direct Memory Access (DMA) engines. The design is then taken through synthesis, implementation, bitstream generation, and hardware export.

3. **Vitis Platform:** The exported hardware is packaged into a platform definition, which captures the processing system configuration, accelerator IP, and I/O interfaces. This platform serves as the target for software applications.

4. **Vitis Application:** Finally, the software application is built on top of the platform, linking the accelerator kernels with host code. The result is deployed to the target board for execution and testing.

Each accelerator core exposes a AXI4-Stream interface data and a AXI4-Lite interface for control. This interface standard enables integration within the block design.

6.2 Hardware Cores

Instead of hard-coding each layer individually (55 layers), a more generic approach was taken. This approach is built around three parameterized cores that together cover all convolutional cases required by *Mobile-CMUNeXt*. This modular approach maximizes reuse across the network, reduces verification effort, and creates a clear mapping between high-level layers and hardware blocks.

The hardware design is organized around three core building blocks:

- **Pointwise (PW) Convolution Core:** Performs 1×1 convolutions that mix information across channels and performs the last layer classification.

- **Depthwise (DW) Convolution Core:** Performs spatial filtering channel by channel (depthwise), with configurable kernel sizes, padding. Also, the core supports optional pooling before the convolution, and supports residual connection addition after the convolution.

- **3D Convolution Core (C3D):** Handles standard convolutions with multiple filters, with support for optional $2\times$ upsampling and a pre-convolution skip connection addition.

All three cores share the same design principles, operating on fixed-point tensors exported from quantization-aware training (with BN-folding applied; see Section 5.2 and Section C.4):

- **AXI Interfaces:** All cores use standardized 64-bit AXI4-Stream interfaces for data input/output and AXI4-Lite for control and configuration. This allows plug-and-play integration into the Vivado block design and simplifies system-level integration.

- **Dataflow Design:** Each core follows a streaming dataflow model [73], where input tensors are read, processed, and written in a pipeline. Internal stages are decoupled using FIFOs and line buffers allowing for concurrent execution and maintaining throughput.

- **Configuration Registers:** A 32-bit configuration parameter is provided through AXI4-Lite to specify runtime parameters such as input/output dimensions, kernel size, number of channels, and scaling factors (more about this in each core implementation). This parameter definition enables the same hardware block to be reusable across multiple layer instances.
- **Data Widths:** The design is specific to quantized fixed-point arithmetic, consistent with the QAT training setup:
  - **Inputs:** 8-bit signed integers.
  - **Weights:** 8-bit signed integers.
  - **Bias:** 16-bit signed integers.
  - **Accumulator:** 24-bit signed arbitrary precision integer, initialized with 16-bit bias.
  - **Outputs:** 8-bit signed integer.
- **Channel Parallelism:** All cores adopt 8-lane parallelism, meaning that 8 channels are processed in parallel at every clock cycle.

6.3 Pointwise Convolution Core

The pointwise (PW) convolution core implements the 1×1 convolution operation, corresponding to a per-pixel dot product between an input feature vector of channel dimension C_{in} and a learned weight matrix mapping to C_{out} output channels. This operation performs channel mixing without altering the spatial resolution of the feature map and requires K^2 fewer operations compared to a convolution with a non-unitary ($K > 1$) kernel. Compared to other cores, the PW core is the least structurally complex, as its single purpose is to execute the pointwise convolution in hardware. The PW core data format is defined as follows:

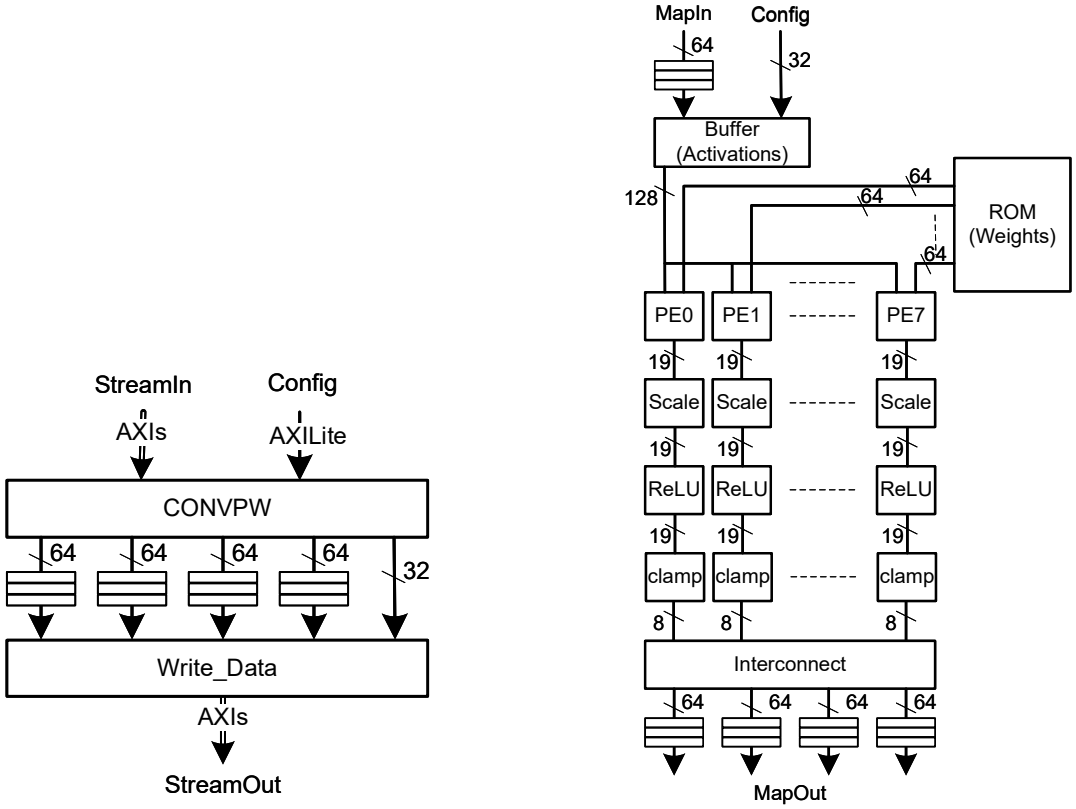
$$\begin{aligned} \textbf{Input:} &\quad [H][W][InC] \\ \textbf{Weights:} &\quad [OutC][InC] \\ \textbf{Bias:} &\quad [OutC] \\ \textbf{Output:} &\quad [H][W][OutC] \end{aligned}$$

where H and W are the spatial dimensions of the feature map, InC is the number of input channels and $OutC$ is the number of output channels.

At the system level, shown in Figure 6.1a, the core is wrapped by AXI4-Stream interfaces, receiving input feature maps *StreamIn* where each 64-bit word packs eight signed 8-bit channels, and receiving the AXLite configuration *Config*. The configuration word defines the input map size, the number of input and output channels groups (so $C_{\text{in}} = 8 \times \texttt{channel}$), and the layer identifier. The number of input words per pixel determines $C_{in} = 8 \times \texttt{channel}$, while the number of output words per pixel sets $C_{out} = 8 \times \texttt{filters}$.

Internally, the core is composed of two modules: the computing engine (`CONVPW`) and the stream writer (`Write_Data`). The computing engine, whose structure is depicted in Figure 6.1b, operates on two rows of the feature map at a time. Input data is buffered locally, and for each column

the engine iterates over output channel groups of eight. Accumulators are initialized with the bias values, and input channels are streamed in 64-bit packed words. Each of the eight lanes inside a word is multiplied in parallel with eight filter weights across two rows, yielding 128 multiply-accumulate operations per cycle in the innermost loop. When accumulation is complete, results are scaled, clamped with ReLU saturation, and packed back into 64-bit words. Outputs are alternately written into two sets of FIFOs in a ping-pong scheme, decoupling the computing and write stages.



(a) Top-level system integration of the Pointwise Convolution core.

(b) Processing element structure of the Pointwise Convolution core.

Figure 6.1: Architecture overview of the Pointwise Convolution core (PW). (a) shows the top-level system integration, while (b) details the internal processing element structure.

The writer module drains the ping-pong FIFOs and generates the output stream, alternating between buffer sets. It ensures alignment with the compute stage and marks the end of the feature map with the TLAST flag. This ping-pong strategy enables overlap between computation and output streaming without stalling. The design exposes parallelism at multiple levels: eight input lanes are processed simultaneously within a word, eight output channels are produced in parallel per filter group, and two rows are computed together. This structured concurrency allows the computing stage to sustain high throughput once the pipeline is filled.

The PW core is the least versatile of all the cores. As shown in Table 6.1, it can be configured for different input-output channel mappings across multiple resolutions, supporting expansion layers (e.g., $8 \rightarrow 32$ channels) and projection layers (e.g., $32 \rightarrow 8$ channels). The complete HLS implementation of the PW core is provided in Section D.1.

Case	Description	Input	Output	Bias	Last
0	Encoder 1 / Decoder 4 — Expansion	256×256×8	256×256×32	32	No
1	Encoder 1 / Decoder 4 — Projection	256×256×32	256×256×8	8	No
2	Encoder 2 / Decoder 3 — Expansion	128×128×8	128×128×32	32	No
3	Encoder 2 / Decoder 3 — Projection	128×128×32	128×128×8	8	No
4	Encoder 3 / Decoder 2 — Expansion	64×64×8	64×64×32	64	No
5	Encoder 3 / Decoder 2 — Projection	64×64×32	64×64×8	16	No
6	Encoder 4 / Decoder 1 — Expansion	32×32×16	32×32×64	64	No
7	Encoder 4 / Decoder 1 — Projection	32×32×64	32×32×16	16	No
8	Bottleneck — Expansion	16×16×16	16×16×64	64	No
9	Bottleneck — Projection	16×16×64	16×16×16	16	No
10	Final Classifier	256×256×8	256×256×1	32	Yes

Table 6.1: PW use cases.

6.4 Depthwise Convolution Core

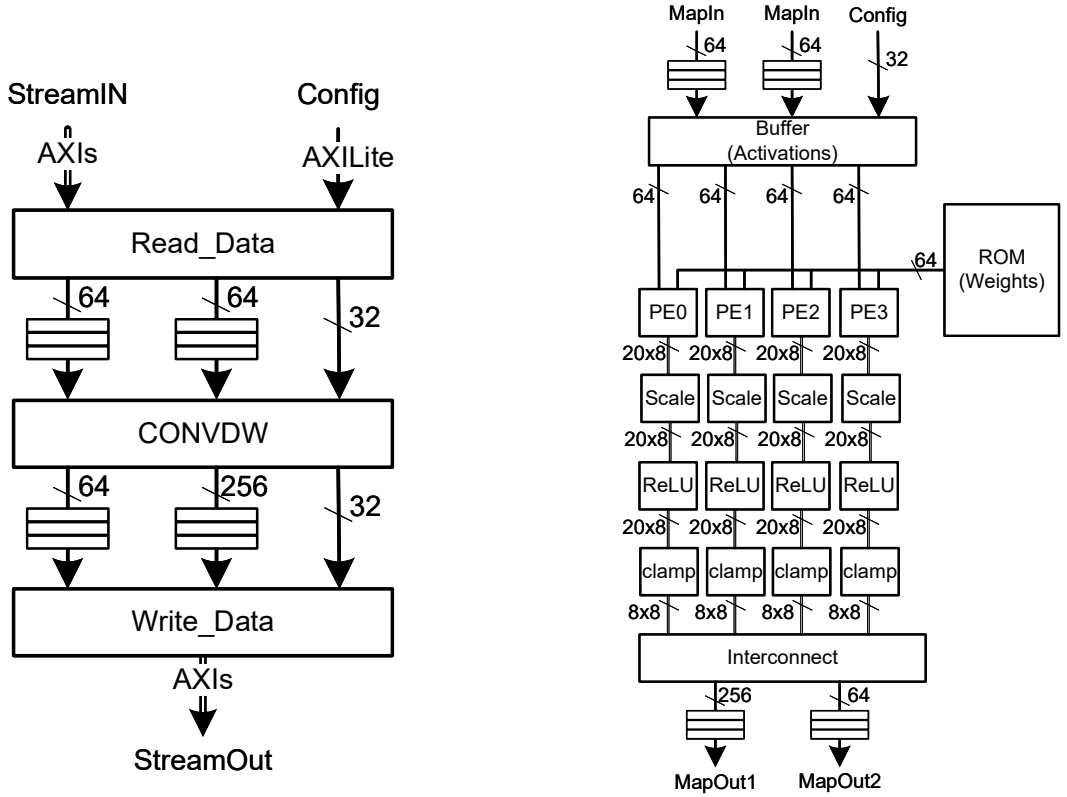
The depthwise (DW) convolution core implements per-channel spatial convolution with optional 2×2 max pooling and an in-place residual connection addition, all in a single streaming pipeline. Inputs arrive as AXI4-Stream 64-bit words, each packing eight signed 8-bit activations for one channel group. The configuration word specifies the map spatial size, the number of input channel groups, kernel size $\in \{3, 7, 9\}$, the padding, the layer identification, and whether to apply $2\times$ downsampling via max pooling. When pooling is enabled, the map processed by convolution is $H/2 \times W/2$, otherwise it is $H \times W$. The output is emitted as 64-bit words on an AXI4-Stream interface, preserving the input packing of eight 8-bit values per word. The DW core data format is defined as follows:

$$\begin{aligned} \textbf{Input:} \quad & [H][W][C] \\ \textbf{Kernel:} \quad & [K_h][K_w][C] \\ \textbf{Bias:} \quad & [C] \\ \textbf{Output:} \quad & [H][W][C] \end{aligned}$$

The block-level datapath of the core is illustrated in Figure 6.2b, while Figure 6.2a shows how the DW core integrates into the accelerator's pipeline. The core has three streaming stages under: `Read_Data` (ingest and pre-processing), `CONVDW` (depthwise spatial MACs), and `Write_Data`. Three FIFOs decouple the rates: `f1` (64-bit) feeds the convolution, `f2` (256-bit) carries vectorized convolution results, and `r_fifo` (64-bit) holds a residual copy of the pre-processed stream for the final residual connection addition.

In the first stage, `Read_Data` writes top/left/right/bottom zero padding into the output stream, so the convolution receives a fully padded tensor, and optionally performs inline 2×2 max pooling before forwarding data. For `maxpool`=0, the function simply forwards input words to both the convolution FIFO and the residual FIFO. For `maxpool`=1, it accumulates two adjacent columns across two consecutive rows using small on-chip line buffers, computes the per-byte maximum

(vectorized across the eight lanes of each 64-bit word), and writes the pooled result to both streams.



(a) Top-level integration of the DW core within the accelerator design.

(b) Block-level architecture of the DW convolution datapath.

Figure 6.2: Architecture overview of the Depthwise (DW) Convolution core. (a) shows the top-level system integration, while (b) details the internal datapath organization.

The second stage, `CONVDW`, performs the spatial convolution per channel group. A sliding window buffer is implemented in LUTRAM to minimize BRAM usage and sustain throughput. Parallelism is exploited at three levels: (i) each 64-bit word carries eight lanes, all processed in parallel, (ii) up to four output columns are computed simultaneously using independent accumulators, and (iii) when multiple channel groups are present, two groups can be processed in parallel. After accumulation, results are scaled, clamped with ReLU saturation, and packed into 256-bit vectors for streaming.

The final stage, `Write_Data`, reconstructs 64-bit words from the 256-bit vectors and performs residual addition with the stream from `r_fifo`. Additions are performed per-byte with symmetric saturation to $[-128, 127]$.

The organization of the **DW core** provides support for the encoder use cases, as summarized in Table 6.2. Common scenarios include lightweight convolutions at high spatial resolutions (e.g., 256×256 with a 3×3 kernel), strong downsampling with larger 7×7 or 9×9 kernels, and max-pooling stages that halve the feature-map resolution. The complete HLS implementation of the DW core is included in Section D.2.

Case	Description	Input	Output	Kernel	Bias	Pad	Maxpool
0	Encoder 1	256×256×8	256×256×8	3×3×8	8	1	No
1	Encoder 2 with Maxpool	256×256×8	128×128×8	3×3×8	8	1	Yes
2	Encoder 2	128×128×8	128×128×8	3×3×8	8	1	No
3	Encoder 3 with Maxpool	128×128×8	64×64×8	7×7×8	8	3	Yes
4	Encoder 3	64×64×8	64×64×8	7×7×8	8	3	No
5	Encoder 4 with Maxpool	64×64×16	32×32×16	7×7×16	16	3	Yes
6	Encoder 4	32×32×16	32×32×16	7×7×16	16	3	No
7	Bottleneck with Maxpool	32×32×16	16×16×16	9×9×16	16	4	Yes
8	Bottleneck	16×16×16	16×16×16	9×9×16	16	4	No

Table 6.2: DW use cases.

6.5 3D Convolution Core

The C3D core implements a streamed 3×3 convolution with optional $2\times$ upsampling and a configurable pre-convolution skip connection addition. The core interface exposes three AXI4-Stream channels: the main input (`Stream1`), an optional skip input (`Stream2`), and the output (`StreamOut`). The configuration word selects the spatial map size, the number of input and output groups, the upsample mode, layer identifier (to index weights/bias), and a flag for the first layer. The C3D core data format is defined as follows:

$$\begin{aligned} \textbf{Input:} &\quad [H][W][inC] \\ \textbf{Kernel:} &\quad [K_h][K_w][outC][inC] \\ \textbf{Bias:} &\quad [outC] \\ \textbf{Output:} &\quad [H][W][outC] \end{aligned}$$

All stages run concurrently and communicate via bounded FIFOs, as shown in the block-level architecture in Figure 6.3b.

The first stage, `Upsample`, prepares the input feature maps before forwarding them to the convolution pipeline. This module supports two operating modes. In the standard mode, it streams the incoming feature map and optionally sums the skip connection stream. When the upsample flag is enabled, however, it performs a $2\times$ bilinear interpolation (see Listing 6.1) in the spatial domain to enlarge the feature-map resolution.

```
procedure BilinearUpsampleFixed(input_image[C][H][W], output_image[C][2H][2W]):
    output_height <- 2 * H
    output_width  <- 2 * W

    for oh from 0 to output_height-1:      // loop over output rows
        iy0 <- floor((oh-1)/2) if oh > 0 else 0
        iy1 <- min(iy0 + 1, H-1)
```

```
8         y_lerp <- 3 if oh is even else 1
9
10        for ow from 0 to output_width-1:   // loop over output columns
11            ix0 <- floor((ow-1)/2) if ow > 0 else 0
12            ix1 <- min(ix0 + 1, W-1)
13            x_lerp <- 3 if ow is even else 1
14
15            for c from 0 to C-1:            // loop over channels
16                // Four corner pixels of interpolation square
17                a <- input_image[c][iy0][ix0]   // top-left
18                b <- input_image[c][iy0][ix1]   // top-right
19                c_ <- input_image[c][iy1][ix0]  // bottom-left
20                d <- input_image[c][iy1][ix1]   // bottom-right
21
22                top    <- a*(4 - x_lerp) + b*x_lerp
23                bottom <- c_*(4 - x_lerp) + d*x_lerp
24                value  <- (top*(4 - y_lerp) + bottom*y_lerp) >> 4
25
26                // Saturate result to int8 range [-128,127]
27                if value > 127 then value <- 127
28                else if value < -128 then value <- -128
29
30                output_image[c][oh][ow] <- value
```

Listing 6.1: Pseudo-code for fixed-point bilinear upsampling.

The interpolation in Listing 6.1 operates per channel (line 15) using fixed-point arithmetic to avoid floating-point multipliers. For each output coordinate (oh, ow) (line 5–line 10), the nearest input pixels $(iy0, ix0)$–$(iy1, ix1)$ are computed (line 6, line 11) to form a 2×2 square of samples (line 17–line 20). Horizontal interpolation is applied first to compute the top and bottom intermediate values (line 22, line 23), followed by vertical interpolation (line 24). The weighting factors alternate between 3 and 1 (line 13, line 8) depending on whether the output index is even or odd, implementing bilinear scaling with integer operations. Finally, the accumulated value is right-shifted by four to normalize the fixed-point scale and saturated to the signed 8-bit range $[-128, 127]$ (line 28) before being written to the output stream (line 30).

The fixed-point kernel emulates the standard floating-point bilinear interpolation, where (a, b, c, d) denote the four corner samples (see Equation 6.1):

$$\begin{aligned} \alpha \in \{0.25,\, 0.75\}, \qquad & \beta \in \{0.25,\, 0.75\}, \\ \text{top} = a(1-\alpha) + b\alpha, \quad & \text{bottom} = c(1-\alpha) + d\alpha, \\ \text{value} = \text{top}(1-\beta) + & \text{bottom}\,\beta. \end{aligned} \tag{6.1}$$

In the fixed-point implementation, the floating-point weights α, β are replaced by integer equivalents scaled by 4:

$$x_{\text{lerp}} \in \{3, 1\}, \quad y_{\text{lerp}} \in \{3, 1\}, \quad \alpha = \tfrac{x_{\text{lerp}}}{4}, \quad \beta = \tfrac{y_{\text{lerp}}}{4}.$$

Even indices correspond to $\alpha = \beta = 0.75$ (closer to the right/bottom sample), while odd indices correspond to 0.25. This design allows fully integer arithmetic while maintaining sub-pixel accuracy.

The integer operations

$$\text{top} = a\,(4 - x_{\text{lerp}}) + b\,x_{\text{lerp}}, \quad \text{bottom} = c\,(4 - x_{\text{lerp}}) + d\,x_{\text{lerp}},$$

followed by

$$\text{value}_{\text{acc}} = \text{top}\,(4 - y_{\text{lerp}}) + \text{bottom}\,y_{\text{lerp}}, \qquad \text{value} = \text{value}_{\text{acc}} \gg 4,$$

are mathematically equivalent to the floating-point formulation, since the right-shift by 4 divides by 16, canceling the scaling introduced horizontally and vertically.

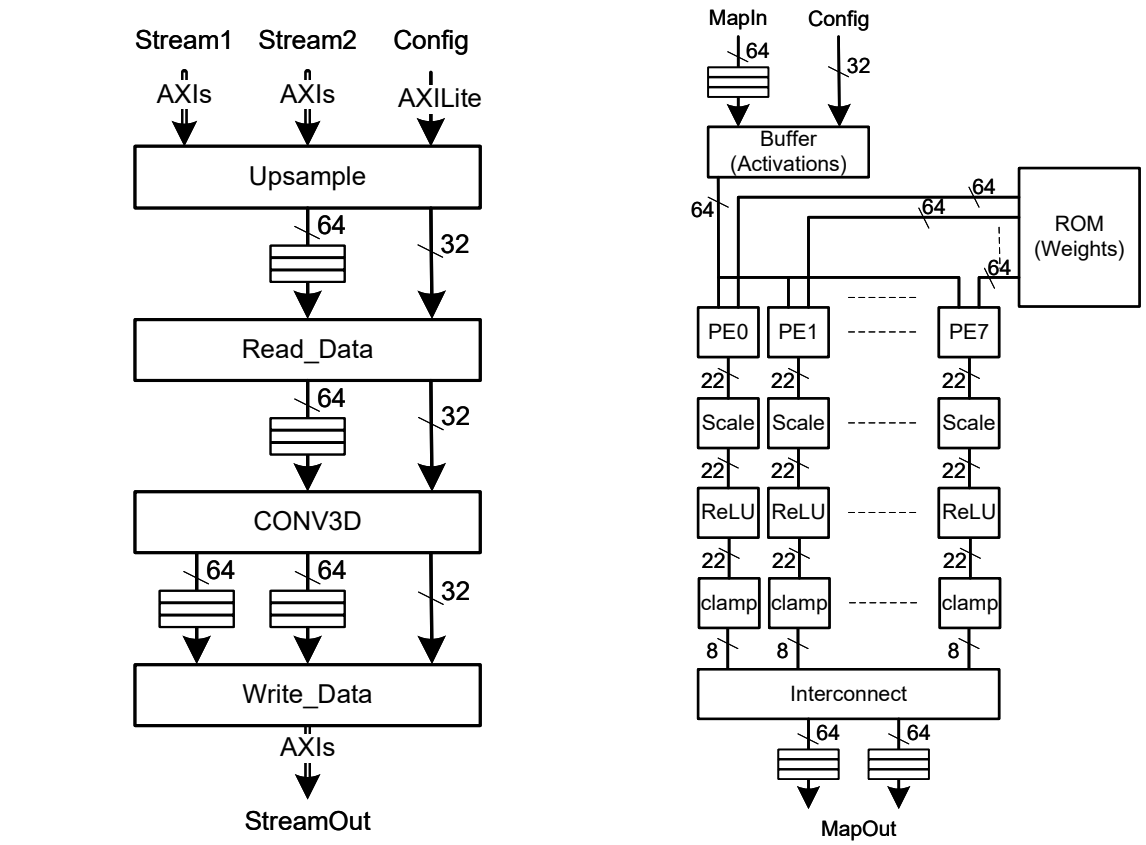
In addition to upsampling, the `Upsample` stage also supports skip connections. When enabled, it performs an element-wise, saturating addition between the incoming stream and the skip stream (`Stream2`). This enables encoder–decoder architectures to fuse high-resolution features from earlier layers with the upsampled activations without off-chip memory transfers.

The padding stage `Read_Data` then standardizes the spatial dimensions for the MAC engine. It computes the output size ($\texttt{map\_size\_up} = \texttt{map\_size} \times (1 + \texttt{upsample})$) and writes a one-pixel zero border around every row, producing a stream that corresponds to $(H{+}2) \times (W{+}2)$ with channels packed as 64-bit words. The padding adapts to the current map size and channel group count.

Next, the computing stage `CONV3D` performs the spatial convolution using a four-line circular window buffer. At each output row it steps the column index by two and computes two adjacent output columns concurrently. For every output group of eight channels, the engine initializes two accumulator vectors with the corresponding bias values. It then iterates over input channel groups and output channel groups, reading two aligned 64-bit words from the line buffer to cover the pair of output columns. The inner loops perform vector and output parallelism: for each of the eight input lanes in a word, products are accumulated into eight outputs simultaneously, yielding a $8{\times}8$ lane-by-output compute tile per column. Multiply operations are bound to DSP blocks, while accumulation is kept in fabric to balance resource pressure and initiation interval. After this both accumulators are right-shifted by the configurable scale, clamped with ReLU saturation, and packed into two 64-bit words that are streamed to separate output FIFOs.

The final stage `Write_Data` drains the two compute FIFOs and formats the final AXI4-Stream. For each output row it reads pairs of 64-bit words across all output channel groups, writing them to `StreamOut`. This two-column production scheme hides address generation and buffering latency inside the compute stage, while the writer sustains a steady downstream rate. The full top-level integration of the C3D core with streaming and AXI control interfaces is illustrated in Figure 6.3a.

The C3D core provides the flexibility required to support diverse deployment scenarios within the accelerator. In encoder layers, it functions as a standard $3{\times}3$ convolution without upsampling. In decoder layers, the core extends its functionality by enabling $2\times$ bilinear upsampling prior to convolution with the option to include skip connections for residual learning. The C3D configurations are listed in Table 6.3, covering the full range of encoder–decoder use cases. The complete HLS implementation of the C3D core is provided in Section D.2.



(a) System integration of the C3D layer with AXI streaming and control interfaces.

(b) Block-level architecture of the C3D core.

Figure 6.3: Architecture overview of the C3D convolution core. (a) illustrates the top-level system integration and interface connectivity, while (b) details the internal datapath and control structure.

Case	Description	Input	Output	Kernel	Bias	Up	Skip
0	Stem	256×256×3	256×256×8	3×3×3×8	8	No	No
1	Encoder 1	256×256×8	256×256×8	3×3×8×8	8	No	No
2	Encoder 2	128×128×8	128×128×8	3×3×8×8	8	No	No
3	Encoder 3	64×64×8	64×64×16	3×3×16×8	16	No	No
4	Encoder 4	32×32×16	32×32×16	3×3×16×16	16	No	No
5	Bottleneck	16×16×16	16×16×24	3×3×24×16	24	No	No
6	Upsample 1	16×16×24	32×32×16	3×3×16×24	16	Yes	No
7	Skip Fusion 1	32×32×16	32×32×16	3×3×16×16	16	No	Yes
8	Upsample 2	32×32×16	64×64×16	3×3×16×16	16	Yes	No
9	Skip Fusion 2	64×64×16	64×64×16	3×3×16×16	16	No	Yes
10	Upsample 3	64×64×16	128×128×8	3×3×8×16	8	Yes	No
11	Skip Fusion 3	128×128×8	128×128×8	3×3×8×8	8	No	Yes
12	Upsample 4	128×128×8	256×256×8	3×3×8×8	8	Yes	No
13	Skip Fusion 4	256×256×8	256×256×8	3×3×8×8	8	No	Yes

Table 6.3: C3D use cases.

6.6 Hardware–Software Integration

The Vitis application running on the ARM Application Processing Unit (APU) orchestrates the execution of convolutional layers by managing data movement and hardware accelerator configuration. Figure 6.4 illustrates the system architecture, where software and PL cooperate through DMA engines and control registers.

6.6.1 System Overview

The architecture is divided between the **processing system (PS)** and the **programmable logic (PL)**. The **APU** runs the software stack, which configures DMA engines, loads input feature maps into DDR memory, and sets parameters for the convolutional accelerators. The **PL** contains three dedicated hardware cores: the 3D convolution (CONV3D), the depthwise convolution (CONVDW), and the pointwise convolution (CONVPW). Each core is connected to the PS through AXI interconnects and has its own DMA interface for high-throughput data transfers. **DDR memory** serves as the global data buffer, holding feature maps, weights, and intermediate results. DMA engines transfer data between DDR and the PL cores without CPU intervention.

6.6.2 Dataflow and Control

The software configures each layer execution by writing parameters (e.g., map size, kernel size, input/output channels, stride, and padding) into the **IP configuration registers** of the corresponding accelerator. Once configured, the workflow proceeds as follows:

1. The APU initializes the **DMA engines**, specifying input and output buffer addresses in DDR.

2. Input feature maps are streamed to the selected convolution core through an AXI-Stream (AXIS) channel.

3. The hardware core performs the convolution, bias addition, activation (ReLU), and optional residual or pooling steps, depending on its type.

4. The output feature maps are streamed back via DMA to DDR memory, ready for either further processing or validation against software implementations.

6.6.3 Parallelism and Modularity

Each convolution type has its own accelerator and DMA interface (DMA0–DMA3), enabling modular testing and integration. Although executions are currently scheduled sequentially by the software, the architecture allows for **parallel data movement and computation**, where one DMA can preload data while another accelerator is computing. This design ensures scalability to more complex pipelines.

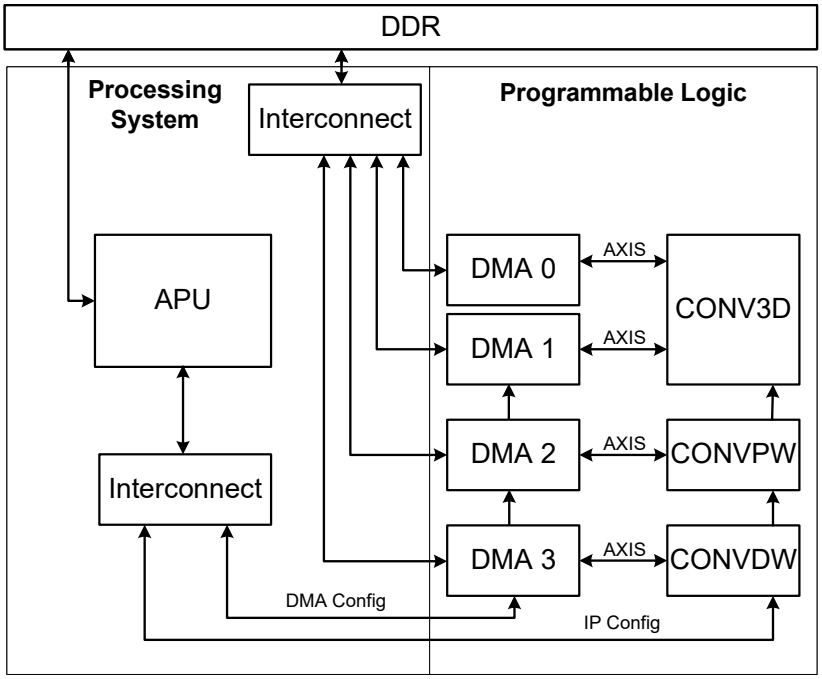
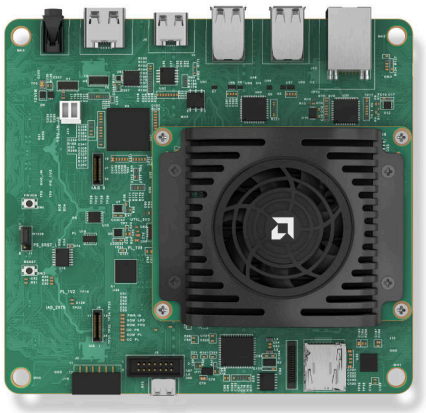


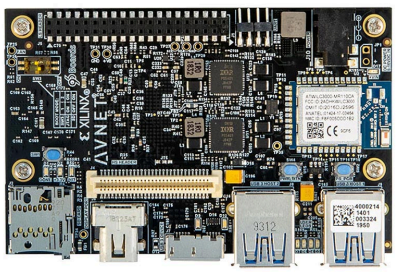
Figure 6.4: Hardware–software architecture: the APU controls DMA engines and convolution accelerators (CONV3D, CONVDW, CONVPW) through AXI interconnects, while DDR memory stores feature maps and weights.

6.7 Choosing Suitable Board

During the initial deployment phase, two candidate FPGA platforms were considered: the **Kria AI Vision Starter Kit (KV260)** [74] (see Figure 6.5a) and the **Ultra96-V2 Avnet** [75] (see Figure 6.5b). Both platforms integrate Zynq UltraScale+ MPSoC devices but differ significantly in terms of available resources. The selection process focused on identifying a device that could sustain the storage and compute requirements of convolutional neural networks, in particular the need to buffer large feature maps and store filter weights and bias terms on-chip.



(a) Kria AI Vision Starter Kit (KV260) board. Copyright 2025, AMD.



(b) Avnet Ultra96-V2 development board. Copyright 2018, AVNET.

Figure 6.5: FPGA platforms. (a) shows the Kria KV260 AI Vision Starter Kit, and (b) shows the Avnet Ultra96-V2 board, both based on the Zynq UltraScale+ MPSoC

The **BRAM blocks** and **DSP slices** are decisive factors. BRAM is essential for storing feature

maps and intermediate data to minimize off-chip memory traffic, while DSP slices are required to implement the multiply-accumulate (MAC) operations that dominate convolutional workloads. A device with limited BRAM would struggle to provide sufficient buffering for inputs, while an insufficient number of DSPs would limit parallelism in convolution execution. Table 6.4 compares the key hardware resources of the two SoC devices.

Resource	Kria KV260	Ultra96-V2
LUTs	117,120	70,560
FFs	234,240	141,120
BRAM (36 Kb)	144	216
URAM (288 Kb)	64	0
DSPs	1,248	360

Table 6.4: Resource comparison between candidate FPGA platforms. Source [76, 77].

In conclusion, while the Kria KV260 features higher overall logic and compute capacity—with significantly more LUTs, FFs, DSPs, and the addition of URAM blocks — that can complement BRAM for large on-chip storage — the Ultra96-V2 Avnet board provides a larger number of BRAM blocks, which are directly beneficial for buffering feature maps and storing intermediate results in convolutional pipelines. Since the proposed accelerator is highly memory-intensive and depends primarily on BRAM bandwidth to maintain throughput, this advantage proved more practical than the additional URAM capacity of the Kria device. Furthermore, the Ultra96-V2 was easier to integrate within the development flow and offered a more compact, convenient form factor for prototyping. For these reasons, the Ultra96-V2 was chosen as the deployment platform.

6.8 Clock Frequency

During high-level synthesis and hardware validation, several clock period constraints were evaluated to determine the optimal operating point for the accelerator cores. The synthesis and co-simulation tests explored a range of timing targets, gradually tightening the period constraint (starting from 1.0 ns)until timing violations began to appear. Through this iterative process, a clock period of **4.0 ns** (corresponding to a frequency of **250 MHz**) was identified as the maximum stable and comfortable operating speed across all modules.

This frequency provided a good balance between performance and implementation margin, ensuring that all processing cores met timing requirements without over-constraining the design or increasing resource utilization. Consequently, **all reported performance and power results in the following sections are based on a target 250 MHz clock frequency**, which serves as the reference operating point for the entire accelerator system.

6.9 Deployment Results

This section reports the experimental outcomes of deploying the accelerator on the chosen board. The evaluation focuses on three aspects: hardware resource utilization, execution la-

tency, and computational throughput. Together, these metrics provide a clear view of how efficiently the architecture maps to the target FPGA fabric and the performance improvements achieved compared to software-only execution.

The subsection on **resource utilization** highlights the distribution of logic, memory, and DSP usage, emphasizing the balance between available resources and architectural requirements. **Execution time measurements** are then presented for different convolutional kernels, comparing hardware-accelerated performance against equivalent software implementations running on the embedded ARM cores. Finally, a detailed **performance analysis** is provided, including multiply-accumulate (MAC) counts and effective throughput in operations per second (OPS), to quantify the acceleration achieved across pointwise (PW), depthwise (DW), and 3D convolution (C3D) layers.

6.9.1 Resource Utilization

The design was synthesized and implemented on the **Ultra96-V2 Avnet** board, which integrates a **Zynq UltraScale+ ZU3EG** device [77]. The post-implementation resource utilization is summarized in Table 6.5.

Module	**LUTs**	**LUT %**	**FFs**	**FF %**	**BRAMs**	**BRAM %**	**DSPs**	**DSP %**
DMA (0–3)	5,648	8.0%	8,660	6.1%	10.5	4.9%	0	0.0%
SMC	1,095	1.6%	1,147	0.8%	0	0.0%	0	0.0%
Ext SMC	4,608	6.5%	6,441	4.6%	0	0.0%	0	0.0%
C3D Core	9,011	12.8%	7,926	5.6%	67	31.0%	151	41.9%
DW Core	10,537	14.9%	4,992	3.5%	42.5	19.7%	35	9.7%
PW Core	8,044	11.4%	4,488	3.2%	48	22.2%	70	19.4%
Total	**38,943**	**55.2%**	**33,654**	**23.8%**	**168**	**77.8%**	**256**	**71.1%**

Table 6.5: Post-implementation FPGA resource utilization per core on the Zynq UltraScale+ ZU3EG device.

The design uses 55.2% of LUTs, 23.8% of flip-flops (FFs), 77.8% of block RAMs (BRAMs), and 71.1% of DSP slices, fitting within the ZU3EG device while leaving margin for future extensions. Resource usage varies across modules, with the control infrastructure (DMA and memory controllers) and the three processing cores contributing differently.

The DMA and Smart Memory Controller (SMC) together consume 9.7% of LUTs and 7.5% of FFs. This usage is associated with data transfer management and memory arbitration. The C3D, depthwise (DW), and pointwise (PW) cores account for most arithmetic and memory resources.

The C3D core is the largest consumer of BRAMs and DSPs, using 31.0% of BRAMs and 41.9% of DSP slices. This is due to the higher number of MACs operations required by 3D convolutions and the need for buffering multiple feature map slices. Additional control logic for skip connections and upsampling also contributes to its resource usage.

The DW core primarily uses logic resources, consuming 14.9% of LUTs and 9.7% of DSPs. This reflects the structure of depthwise convolutions, where computation is distributed across channels with limited arithmetic intensity. The PW core shows a more balanced profile, using 11.4% of LUTs and 19.4% of DSPs, consistent with its role in channel mixing.

BRAMs and DSPs are the main limiting resources in this design. BRAM usage is driven by buffering requirements for feature maps, weights, and intermediate results, while DSP usage is dominated by MAC operations. Despite this, sufficient margin remains to support extensions such as additional layers, increased parallelism, or deeper buffering, indicating that the architecture is well aligned with the resources of the ZU3EG device.

6.9.2 Power Efficiency

The power efficiency of the implemented accelerator was evaluated using the Vivado power analysis tool applied to the post-implementation netlist. The analysis accounts for both static and dynamic power components, as well as environmental and thermal conditions corresponding to the target platform. The estimation assumes an ambient temperature of $25.0^{\circ}\mathrm{C}$, a medium-profile heat sink, and an effective thermal resistance of $\theta_{\mathrm{JA}} = 2.7^{\circ}\mathrm{C/W}$. Under these operating conditions, the estimated junction temperature was $36.4^{\circ}\mathrm{C}$, yielding a thermal margin of $63.6^{\circ}\mathrm{C}$ and confirming that the system operates well within safe thermal limits.

The total on-chip power consumption was estimated at

$$P_{\text{total}} = 4.179 \text{ W},$$

of which

$$P_{\text{dynamic}} = 3.847 \text{ W} \quad (92\%), \qquad P_{\text{static}} = 0.332 \text{ W} \quad (8\%).$$

Within the dynamic component, the processing system (PS) represents the largest contributor, consuming **1.418 W** (36% of total on-chip power). The remaining programmable logic (PL) power is distributed among clocks (**0.219 W**, 6%), signals (**0.730 W**, 19%), logic (**0.580 W**, 15%), block RAMs (**0.487 W**, 13%), and DSP slices (**0.412 W**, 11%). The PL static power was estimated at **0.228 W**, while the PS contributes **0.104 W** to the static component.

To provide context, inference workloads executed on general-purpose CPUs typically operate within a power envelope ranging from tens of watts for mobile or embedded processors to over 100 W for server-class CPUs. Similarly, GPU-based inference — while offering significantly higher peak throughput — generally incurs power consumption on the order of 100–500 W, depending on the device class and utilization. In contrast, the proposed accelerator achieves inference at a total on-chip power of approximately 4.18 W, representing a reduction of more than an order of magnitude in power consumption relative to conventional CPU-based inference, and two orders of magnitude relative to GPUs.

This difference highlights the advantage of acceleration on reconfigurable hardware, where computation, memory access, and data movement are tightly coupled and optimized for the target workload. While CPUs and GPUs offer superior flexibility and peak performance, the presented accelerator demonstrates a highly favorable power-efficiency profile.

6.9.3 Execution Time

The execution time measurements summarized in Table 6.6, Table 6.7, and Table 6.8 demonstrate the substantial runtime reductions achieved by the hardware accelerator compared to C/C++ software implementation execution on the same processing system. Across all layer types — Pointwise (PW), Depthwise (DW), and 3D Convolution (C3D) — the hardware consistently completes operations in the microsecond range, while software runtimes extend into hundreds of milliseconds or even seconds.

For the PW cases (Table 6.6), the hardware achieves speedups ranging from $98\times$ up to $259\times$ in most configurations. Case 0 completes in $6.73\ \mathrm{ms}$ on hardware compared to $660.42\ \mathrm{ms}$ in software, resulting in a $98\times$ reduction in latency. Mid-range cases such as Case 6 and Case 7 reach even higher accelerations of $170\times$ and $259\times$, respectively, highlighting the scalability of the pointwise convolution engine across varying input dimensions and channel counts.

Case	HW Cycles	HW Time (μs)	SW Time (μs)	Speedup ($\times$)
0	672 795	6 727.9	660 415.4	98
1	391 461	3 914.6	576 017.2	147
2	165 422	1 654.2	165 110.2	100
3	95 876	958.7	144 012.6	150
4	41 138	411.3	41 282.4	100
5	23 442	234.4	36 002.0	154
6	22 129	221.2	37 630.1	170
7	13 524	135.2	35 006.5	259
8	5 872	58.7	9 406.6	160
9	3 510	35.1	8 751.4	249
10	210 514	2 105.1	22 185.7	11

Table 6.6: Measured execution time for hardware and software PW layers.

For the DW cases (Table 6.7), the hardware achieves speedups between $149\times$ and $292\times$. Case 0 completes in $2.46\ \mathrm{ms}$ in hardware versus $371.01\ \mathrm{ms}$ in software, while Cases 3 and 4 reach the highest accelerations of $289\times$ and $292\times$, respectively. These results confirm the efficiency of the depthwise convolution engine, which leverages high data reuse and spatial parallelism to minimize memory bandwidth demands.

The C3D cases (Table 6.8) represent the most computationally demanding workloads and achieve the highest speedups overall, ranging from $398\times$ to $777\times$. Case 0 executes in $5.44\ \mathrm{ms}$ on hardware compared to $2.17\ \mathrm{s}$ in software, while Case 6 attains the largest speedup of $777\times$. Even the deepest layers, such as Cases 12 and 13, maintain sub-$6\ \mathrm{ms}$ runtimes, confirming that the pipelined and dataflow-oriented architecture efficiently handles the large number of 3D MAC operations.

Overall, the results confirm that the hardware accelerator achieves consistent and substantial performance gains across all tested cases. Average speedups reach approximately $150\times$ for PW, $230\times$ for DW, and over $500\times$ for C3D, demonstrating that the proposed design effectively

Case	HW Cycles	HW Time (μs)	SW Time (μs)	Speedup ($\times$)
0	245 697	2 456.9	371 014.3	151
1	65 539	655.3	102 324.5	156
2	62 233	622.3	92 493.7	149
3	33 932	339.3	98 186.2	289
4	32 695	326.9	95 591.0	292
5	17 595	175.9	46 951.0	267
6	16 698	166.9	45 742.9	274
7	6 700	67.0	16 010.1	238
8	6 293	62.9	15 720.4	250

Table 6.7: DW runtime comparison between hardware and software.

Case	HW Cycles	HW Time (μs)	SW Time (μs)	Speedup ($\times$)
0	543 974	5 439.7	2 173 332.0	400
1	545 674	5 456.7	2 172 951.7	398
2	137 114	1 371.1	543 206.6	396
3	65 205	652.0	270 966.9	416
4	20 841	208.4	129 679.8	622
5	7 950	79.5	48 623.6	612
6	25 022	250.2	194 427.3	777
7	21 180	211.8	130 886.7	618
8	81 700	817.0	526 267.2	644
9	82 486	824.8	523 533.0	635
10	172 991	1 729.9	1 068 983.9	618
11	137 938	1 379.3	552 898.8	401
12	543 554	5 435.5	2 236 647.6	411
13	549 421	5 494.2	2 211 529.0	403

Table 6.8: Execution time comparison between hardware and software implementations of C3D.

exploits parallelism, pipelining, and data reuse. These improvements make the architecture highly suitable for real-time inference workloads under the fixed 250 MHz operating frequency.

6.9.4 Performance

The measured throughput results summarized in Table 6.9, Table 6.10, and Table 6.11 demonstrate that the accelerator sustains high operation rates across all convolution core types. Performance is reported in GOPS, calculated from the total number of multiply–accumulate (MAC) operations and the hardware execution time at a 250 MHz clock frequency.

For the PW convolution cases (Table 6.9), throughput values range between 4.99 and 19.40 GOPS depending on the configuration. The highest performance is achieved in Case 7, which reaches 19.40 GOPS for a 32×32 input and $64 \rightarrow 16$ channel mapping. This performance variation is primarily determined by the degree of parallelism and the ratio between input and output channels. Configurations with higher output channel counts per input feature map better utilize the datapath, while smaller mappings underutilize the available compute units. Nevertheless, all PW cases sustain multi-GOPS throughput, confirming that the channel-mixing engine efficiently leverages the underlying MAC array.

Case	Input	Cin→Cout	MACs (M)	HW Cycles	HW GOPS
0	256×256	8→32	16.78	672 795	6.23
1	256×256	32→8	16.78	391 461	10.72
2	128×128	8→32	4.19	165 422	6.33
3	128×128	32→8	4.19	95 876	10.93
4	64×64	8→32	1.05	41 138	6.38
5	64×64	32→8	1.05	23 442	11.19
6	32×32	16→64	1.05	22 129	11.86
7	32×32	64→16	1.05	13 524	19.40
8	16×16	16→64	0.26	5 872	11.07
9	16×16	64→16	0.26	3 510	18.52
10	256×256	8→1	2.10	210 514	4.99

Table 6.9: Performance of PW convolution cases in terms of MAC count and achieved throughput at 250 MHz.

For the DW convolution cases (Table 6.10), throughput ranges from 4.50 to 13.12 GOPS. Larger spatial resolutions such as Case 0 (256×256) sustain 4.8 GOPS, while smaller configurations like Case 8 (16×16, $K=9$) achieve the peak of 13.12 GOPS. This improvement at reduced spatial sizes is due to increased kernel reuse and reduced control overhead, which allows the datapath to operate closer to its fully unrolled capacity. Across all DW configurations, performance remains stable and well-balanced relative to arithmetic intensity and on-chip memory utilization.

The C3D convolution cases (Table 6.11) exhibit the highest sustained throughput, with performance values between 6.65 and 28.30 GOPS. Early encoder cases (e.g., Case 1 and Case 2) maintain around 17 GOPS, while mid-network stages such as Case 4 and Case 5 peak above

Case	Input	K	Cin	MACs (G)	HW Cycles	HW Time (μs)	HW GOPS
0	256×256	3	8	4.72	245 697	982.8	4.80
1	128×128	3	8	1.18	65 539	262.2	4.50
2	128×128	3	8	1.18	62 233	248.9	4.74
3	64×64	7	8	1.61	33 932	135.7	11.86
4	64×64	7	8	1.61	32 695	130.8	12.31
5	32×32	7	16	0.80	17 595	70.4	11.35
6	32×32	7	16	0.80	16 698	66.8	11.98
7	16×16	9	16	0.33	6 700	26.8	12.31
8	16×16	9	16	0.33	6 293	25.2	13.12

Table 6.10: Performance of DW convolution cases in terms of MAC count and throughput at 250 MHz.

27 GOPS. The decoder stages show similar stability, with Cases 10 through 13 ranging from 17 to 27 GOPS depending on spatial scaling. The relatively flat performance trend across a wide range of input and output sizes indicates that the column-tiling and vector-unrolling strategies amortize memory and control overheads, maintaining high utilization even under asymmetric configurations.

Case	Input→Output	Cin→Cout	K	MACs (M)	HW Cycles	HW GOPS
0	256×256×3 → 256×256×8	3→8	3	14.45	543 974	6.65
1	256×256×8 → 256×256×8	8→8	3	37.75	545 674	17.29
2	128×128×8 → 128×128×8	8→8	3	9.44	137 114	17.22
3	64×64×8 → 64×64×16	8→16	3	4.72	65 205	18.09
4	32×32×16 → 32×32×16	16→16	3	2.36	20 841	28.30
5	16×16×16 → 16×16×24	16→24	3	0.88	7 950	27.69
6	16×16×24 → 32×32×16	24→16	3	1.18	25 022	11.79
7	32×32×16 → 32×32×16	16→16	3	0.79	21 180	9.34
8	32×32×16 → 64×64×16	16→16	3	3.77	81 700	11.54
9	64×64×16 → 64×64×16	16→16	3	9.44	82 486	21.91
10	64×64×16 → 128×128×8	16→8	3	18.87	172 991	27.26
11	128×128×8 → 128×128×8	8→8	3	9.44	137 938	17.12
12	128×128×8 → 256×256×8	8→8	3	37.75	543 554	17.35
13	256×256×8 → 256×256×8	8→8	3	37.75	549 421	17.16

Table 6.11: Performance of C3D convolution cases in terms of MAC count and throughput at 250 MHz.

Overall, the throughput analysis confirms that all three cores sustain multi-GOPS operation across a broad range of input configurations. The PW cores achieve between 5–19 GOPS, the DW cores maintain 4–13 GOPS, and the C3D cores reach up to 28 GOPS. These results demonstrate how the heterogeneous architecture leverages each core's strengths: PW cores for efficient channel mixing, DW cores for low-cost spatial filtering, and C3D cores for compute-intensive 3D feature extraction and decoding. Together, they enable the system to sustain

consistent multi-GOPS throughput at a fixed 250 MHz operating frequency.

6.9.5 Inference Speed

The overall inference speed of the accelerator was evaluated by comparing the average end-to-end latency between the hardware and C/C++ software implementations. Across all test cases, the average software inference time was measured at $22.3\ \mathrm{s}$ per frame, while the hardware implementation achieved an average latency of $100.3\ \mathrm{ms}$ ($0.1003\ \mathrm{s}$) per frame when operating at the nominal clock frequency of 250 MHz.

The corresponding throughput, expressed in frames per second (FPS), is calculated as

$$\mathrm{FPS} = \frac{1}{t_{\mathrm{frame}}}.$$

Hence, the C/C++ software execution achieves only

$$\mathrm{FPS}_{\mathrm{SW}} = \frac{1}{22.5} \approx 0.044 \text{ frames/s},$$

whereas the hardware accelerator reaches

$$\mathrm{FPS}_{\mathrm{HW}} = \frac{1}{0.1003} \approx 9.97 \text{ frames/s}.$$

These results correspond to an improvement of over **225**$\times$ in inference speed. In practical terms, the hardware design is capable of processing approximately ten frames per second, which lies within the lower bound of real-time operation for computer vision workloads.

Overall, these results indicate that the proposed hardware implementation is well suited for embedded real-time inference tasks, achieving near–real-time frame rates while consuming only a fraction of the energy and time required by software execution on the embedded processing system.

6.9.6 Hardware Inference Results

The hardware inference results are shown in Figure 6.6, comparing the input retinal images (top row), ground-truth vessel masks (middle), and predictions from the accelerator (bottom). The hardware model mostly reconstructs the main vascular structure, capturing the overall vessel topology and major arteries visible in the reference masks. Some noise is present in the predictions, with small false detections and missing vessels, especially near thin branches. These artifacts likely stem from quantization effects and small differences between the Brevitas software model and the synthesized HLS implementation. In particular, the fixed-point bilinear upsampling and binary operations in hardware differ slightly from Brevitas's floating-point behavior (see Listing 6.1), causing minor variations that accumulate across layers.

Despite this, the segmentation quality remains visually consistent and sometimes acceptable. The accelerator maintains nearly 10 FPS at 250 MHz, offering a strong trade-off between accuracy, latency, and resource efficiency.

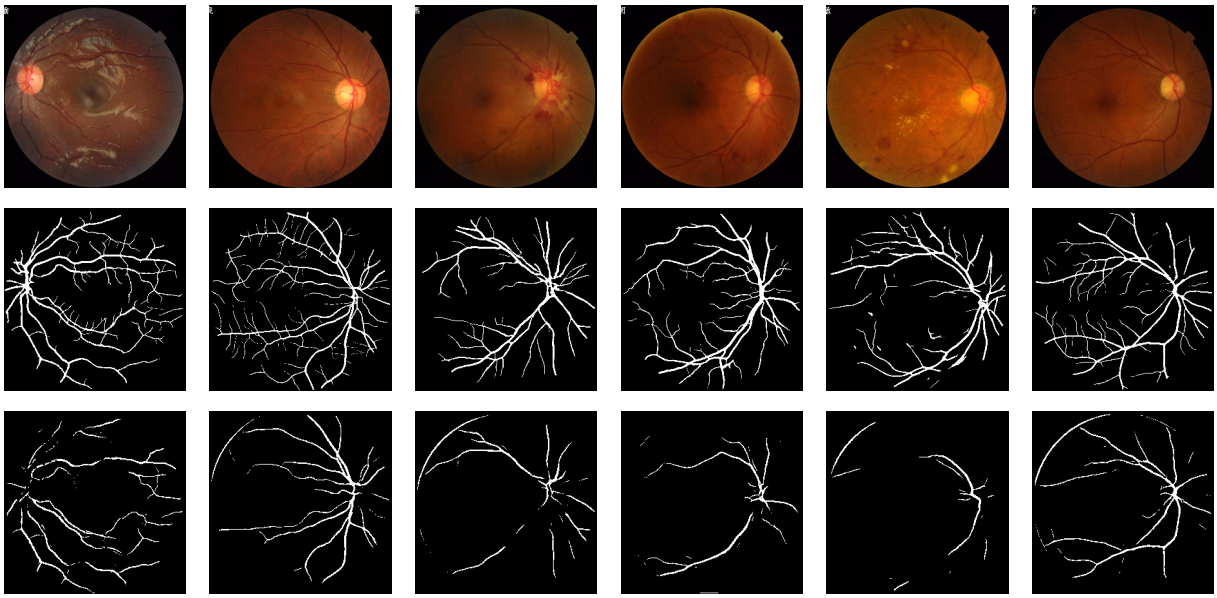


Figure 6.6: Qualitative comparison between input retinal images (top), ground-truth vessel masks (middle), and hardware-generated predictions (bottom). The hardware model correctly identifies major vessels but introduces minor noise and missing vessel segments.

7 Conclusion

This work presented the complete design, optimization, and hardware realization of **Mobile-CMUNeXt**, a quantized convolutional neural network for real-time retinal vessel segmentation on FPGA devices. The research addressed the growing need for low-power, high-performance medical image segmentation at the edge, where data privacy and latency constraints prohibit cloud-based inference.

7.1 Discussion

Starting from an analysis of existing segmentation architectures, the CMUNeXt model was selected as a foundation for its modularity and balanced performance. It was then redesigned into a hardware suitable variant through targeted modifications: simplified activation functions, optimized skip connections, and structural refinements that preserve accuracy while reducing computational cost. Quantization-aware training and batch-normalization folding were applied to enable fixed-point deployment with minimal accuracy loss, confirming that effective medical segmentation can be achieved under integer arithmetic.

On the hardware side, dedicated accelerator cores were implemented for pointwise (PW), depthwise (DW), and 3D (C3D) convolutions. These cores were integrated into a unified inference pipeline synthesized for the **Ultra96-V2 Avnet** board. Although the Kria KV260 was also evaluated as a potential target, the Ultra96-V2 provided a more compact form factor and a higher BRAM count, which proved advantageous for feature-map buffering and on-chip data reuse. Post-synthesis analysis confirmed efficient resource utilization, with balanced LUT, DSP, and BRAM usage and operation at a stable **250 MHz** clock frequency.

The benchmark results demonstrated significant performance gains across all convolutional layers. The accelerator achieved multi-GOPS throughput, reducing execution times from seconds in C/C++ software to milliseconds in hardware — equivalent to speedups exceeding $400\times$ in complex 3D convolution stages. The resulting end-to-end inference reached approximately **10 FPS**, within real-time operation requirements for medical applications.

Qualitative evaluation of hardware-generated segmentation masks showed that the accelerator identifies the main retinal vessels and somewhat preserves the overall vascular structure. Prediction noise and missing fine vessels were observed, likely caused by differences between Brevitas' floating-point upsampling and the fixed-point bilinear interpolation implemented in HLS. Despite this, the visual and quantitative results confirm that the fixed-point hardware model

retains clinically relevant accuracy. In summary, these results demonstrate that FPGA-based accelerators can deliver efficient, low-latency deep learning inference for medical imaging, establishing a viable foundation for practical, deployable edge-AI systems in healthcare.

7.2 Future Work

Future improvements will focus on three main directions. First, the slight deviations observed between PyTorch and hardware accelerated predictions warrant a deeper analysis of fixed-point arithmetic, particularly in the upsampling and activation units, to reduce quantization noise and improve consistency. Second, integrating a more efficient communication interface — transitioning from the current UART link to a USB or Ethernet-based pipeline — will enable faster image transfer and higher throughput for larger datasets. Finally, expanding the accelerator to support higher resolutions and additional medical modalities (e.g., OCT or ultrasound) could extend its clinical applicability. Together, these developments will enhance both the precision and usability of *Mobile-CMUNeXt*, reinforcing the role of FPGA-based inference as a practical, energy-efficient solution for edge medical imaging.

Bibliography

[1] Carsten Maple et al. “The AI Revolution: Opportunities and Challenges for the Finance Sector”. In: *arXiv* (2023). eprint: 2308.16538. URL: https://arxiv.org/abs/2308.16538.

[2] Vladimir Franki, Darin Majnarić, and Alfredo Višković. “A Comprehensive Review of Artificial Intelligence (AI) Companies in the Power Sector”. In: *Energies* 16.3 (2023), p. 1077. DOI: 10.3390/en16031077.

[3] Emma Martinho-Truswell. “How AI Could Help the Public Sector”. In: *Harvard Business Review* 29 (2018). URL: https://hbr.org/2018/01/how-ai-could-help-the-public-sector.

[4] Fei Wang and Anita Preininger. “AI in Health: State of the Art, Challenges, and Future Directions”. In: *Yearbook of Medical Informatics* 28.01 (2019), pp. 016–026. DOI: 10.1055/s-0039-1677908.

[5] Shiva Maleki Varnosfaderani and Mohamad Forouzanfar. “The Role of AI in Hospitals and Clinics: Transforming Healthcare in the 21st Century”. In: *Bioengineering* 11.4 (2024), p. 337. DOI: 10.3390/bioengineering11040337.

[6] S. Niyas et al. “Medical Image Segmentation with 3D Convolutional Neural Networks: A Survey”. In: *Neurocomputing* 493 (2022), pp. 397–413. DOI: 10.1016/j.neucom.2022.04.065.

[7] Eric Topol. *Deep Medicine: How Artificial Intelligence Can Make Healthcare Human Again*. Hachette UK, 2019. ISBN: 9781541644632.

[8] Saeid Asgari Taghanaki et al. “Deep Semantic Segmentation of Natural and Medical Images: A Review”. In: *Artificial Intelligence Review* 54 (2021), pp. 137–178. DOI: 10.1007/s10462-020-09854-1.

[9] Lequan Yu et al. “Automatic 3D Cardiovascular MR Segmentation with Densely-Connected Volumetric ConvNets”. In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 2017, pp. 287–295. DOI: 10.1007/978-3-319-66185-8_33.

[10] Andriy Myronenko. “3D MRI Brain Tumor Segmentation Using Autoencoder Regularization”. In: *International MICCAI BrainLesion Workshop*. Springer, 2018, pp. 311–320. DOI: 10.1007/978-3-030-11726-9_28.

[11] Yaopeng Peng, Milan Sonka, and Danny Z. Chen. “U-Net v2: Rethinking the Skip Connections of U-Net for Medical Image Segmentation”. In: *arXiv* (2023). eprint: 2311.17791. URL: https://arxiv.org/abs/2311.17791.

[12] Samra Irshad, Douglas P. S. Gomes, and Seong Tae Kim. “Improved Abdominal Multi-Organ Segmentation via 3D Boundary-Constrained Deep Neural Networks”. In: *IEEE Access* 11 (2023), pp. 35097–35110. DOI: 10.1109/ACCESS.2023.3264582.

[13] Yiqing Wang et al. “SwinMM: Masked Multi-view with Swin Transformers for 3D Medical Image Segmentation”. In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 2023, pp. 486–496. DOI: 10.1007/978-3-031-43898-1_47.

[14] Ange Lou, Shuyue Guan, and Murray Loew. “CFPNet-M: A Light-weight Encoder-Decoder Based Network for Multimodal Biomedical Image Real-Time Segmentation”. In: *Computers in Biology and Medicine* 154 (2023), p. 106579. DOI: 10.1016/j.compbiomed.2023.106579.

[15] Risheng Wang et al. “Medical Image Segmentation Using Deep Learning: A Survey”. In: *IET Image Processing* 16.5 (2022), pp. 1243–1267. DOI: 10.1049/ipr2.12419.

[16] Konstantinos Liopyris et al. “Artificial Intelligence in Dermatology: Challenges and Perspectives”. In: *Dermatology and Therapy* 12.12 (2022), pp. 2637–2651. DOI: 10.1007/s13555-022-00833-8.

[17] Siming Bayer et al. “Intraoperative Imaging Modalities and Compensation for Brain Shift in Tumor Resection Surgery”. In: *International Journal of Biomedical Imaging* 2017 (2017). Comprehensive review of 126 papers on brain shift causes, measurement, and compensation techniques, p. 6028645. DOI: 10.1155/2017/6028645.

[18] R. M. Comeau et al. “Intraoperative ultrasound for guidance and tissue shift correction in image-guided neurosurgery”. In: *Medical Physics* 27.4 (2000), pp. 787–800. DOI: 10.1118/1.598942.

[19] François-Xavier Carton et al. “Automatic segmentation of brain tumor resections in intraoperative ultrasound images using U-Net”. In: *Journal of Medical Imaging* 7.3 (2020), p. 031503. DOI: 10.1117/1.JMI.7.3.031503.

[20] Santiago Cepeda et al. “Real-time brain tumor detection in intraoperative ultrasound: From model training to deployment in the operating room”. In: *Computers in Biology and Medicine* 193 (2025), p. 110481. DOI: 10.1016/j.compbiomed.2025.110481.

[21] Ahmed Sanaullah et al. “Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks”. In: *BMC Bioinformatics* 19.Suppl 18 (2018), p. 490. DOI: 10.1186/s12859-018-2505-7.

[22] Ahmed Sanaullah et al. “Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks”. In: *BMC Bioinformatics* 19.Suppl 18 (2018), p. 490. DOI: 10.1186/s12859-018-2505-7.

[23] Siyu Xiong et al. “MRI-based brain tumor segmentation using FPGA-accelerated neural network”. In: *BMC Bioinformatics* 22.1 (2021), p. 421. DOI: 10.1186/s12859-021-04347-6.

[24] Mrinalini Joshi-Pangaonkar et al. “FPGA Based Image Segmentation for Medical Image Analysis for Disease Diagnosis”. In: *Proceeding of the 1st International Conference on Lifespan Innovation (ICLI 2025)*. Vol. 88. Advances in Health Sciences Research. Atlantis Press, 2025, pp. 298–304. DOI: 10.2991/978-94-6463-831-8_36.

[25] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 25. 2012, pp. 1097–1105. URL: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[27] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision*

and Pattern Recognition (CVPR). IEEE, 2015, pp. 3431–3440. DOI: 10.1109/CVPR.2015.7298965.

[28] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. "SegNet: A Deep Convolutional Encoder–Decoder Architecture for Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (2017), pp. 2481–2495. DOI: 10.1109/TPAMI.2016.2644615.

[29] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv* (2014). eprint: 1409.1556. URL: https://arxiv.org/abs/1409.1556.

[30] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[31] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. "V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation". In: *2016 Fourth International Conference on 3D Vision (3DV)*. IEEE, 2016, pp. 565–571. DOI: 10.1109/3DV.2016.79.

[32] Liang-Chieh Chen et al. "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40.4 (2018), pp. 834–848. DOI: 10.1109/TPAMI.2017.2699184.

[33] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *arXiv* (2020). Presented at ICLR 2021 (OpenReview). arXiv: 2010.11929 [cs.CV]. URL: https://arxiv.org/abs/2010.11929.

[34] Ilya O. Tolstikhin et al. "MLP-Mixer: An All-MLP Architecture for Vision". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 34. Curran Associates, Inc., 2021, pp. 24261–24272. URL: https://proceedings.neurips.cc/paper/2021/hash/cba062786f4f0c92bde5f16c3e2a12a2-Abstract.html.

[35] Dongze Lian et al. "AS-MLP: An Axial Shifted MLP Architecture for Vision". In: *arXiv* (2021). arXiv: 2107.08391. URL: https://arxiv.org/abs/2107.08391.

[36] Hugo Touvron et al. "ResMLP: Feedforward Networks for Image Classification with Data-Efficient Training". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.4 (2023), pp. 5314–5321. DOI: 10.1109/TPAMI.2022.3155635.

[37] Tan Yu et al. "S^2-MLP: Spatial-Shift MLP Architecture for Vision". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2022, pp. 297–306. DOI: 10.1109/WACV51458.2022.00036.

[38] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.

[39] Zongwei Zhou et al. "UNet++: A Nested U-Net Architecture for Medical Image Segmentation". In: *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support (DLMIA/ML-CDS)*. Springer, 2018, pp. 3–11. DOI: 10.1007/978-3-030-00889-5_1.

[40] Ozan Oktay et al. "Attention U-Net: Learning Where to Look for the Pancreas". In: *arXiv* (2018). eprint: 1804.03999. URL: https://arxiv.org/abs/1804.03999.

[41] Jieneng Chen et al. “TransUNet: Transformers Make Strong Encoders for Medical Image Segmentation”. In: *arXiv* (2021). eprint: 2102.04306. URL: https://arxiv.org/abs/2102.04306.

[42] Hu Cao et al. “Swin-Unet: Unet-like Pure Transformer for Medical Image Segmentation”. In: *European Conference on Computer Vision (ECCV)*. Springer, 2022, pp. 205–218. DOI: 10.1007/978-3-031-25066-8_9.

[43] Hanguang Xiao et al. “Transformers in Medical Image Segmentation: A Review”. In: *Biomedical Signal Processing and Control* 84 (2023), p. 104791. DOI: 10.1016/j.bspc.2023.104791.

[44] Binh-Duong Dinh et al. “1M Parameters are Enough? A Lightweight CNN-based Model for Medical Image Segmentation”. In: *2023 Asia Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 2023, pp. 1279–1284. DOI: 10.1109/APSIPAASC58517.2023.10317244.

[45] Resha Dwika Hefni Al-Fahsi et al. “GIVTED-Net: GhostNet-Mobile Involution ViT Encoder-Decoder Network for Lightweight Medical Image Segmentation”. In: *IEEE Access* 12 (2024), pp. 81281–81292. DOI: 10.1109/ACCESS.2024.3411870.

[46] Kai Han et al. “GhostNet: More Features from Cheap Operations”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2020, pp. 1580–1589. DOI: 10.1109/CVPR42600.2020.00165.

[47] Duo Li et al. “Involution: Inverting the Inherence of Convolution for Visual Recognition”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2021, pp. 12321–12330. DOI: 10.1109/CVPR46437.2021.01214.

[48] Qingshan She et al. “LUCF-Net: Lightweight U-Shaped Cascade Fusion Network for Medical Image Segmentation”. In: *IEEE Journal of Biomedical and Health Informatics* (2024). DOI: 10.1109/JBHI.2024.3506829.

[49] Fenghe Tang et al. “CMUNeXt: An Efficient Medical Image Segmentation Network Based on Large Kernel and Skip Fusion”. In: *2024 IEEE International Symposium on Biomedical Imaging (ISBI)*. IEEE, 2024, pp. 1–5. DOI: 10.1109/ISBI56570.2024.10635609.

[50] Zhimeng Han, Muwei Jian, and Gai-Ge Wang. “ConvUNeXt: An Efficient Convolution Neural Network for Medical Image Segmentation”. In: *Knowledge-Based Systems* 253 (2022), p. 109512. DOI: 10.1016/j.knosys.2022.109512.

[51] Fenghe Tang et al. “CMU-Net: A Strong ConvMixer-Based Medical Ultrasound Image Segmentation Network”. In: *2023 IEEE 20th International Symposium on Biomedical Imaging (ISBI)*. IEEE, 2023, pp. 1–5. DOI: 10.1109/ISBI53787.2023.10230609.

[52] Hardik Sharma et al. “From High-Level Deep Neural Models to FPGAs”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783720.

[53] Ye Liu et al. “A Fast and Efficient FPGA-Based Level Set Hardware Accelerator for Image Segmentation”. In: *2020 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. IEEE, 2020, pp. 61–62. DOI: 10.1109/ICTA50426.2020.9331957.

[54] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *arXiv* (2017). arXiv: 1704.04861 [cs.CV]. URL: https://arxiv.org/abs/1704.04861.

[55] Yanru Lin, Yanjun Zhang, and Xu Yang. “A Low Memory Requirement MobileNets Accelerator Based on FPGA for Auxiliary Medical Tasks”. In: *Bioengineering* 10.1 (2022), p. 28. DOI: 10.3390/bioengineering10010028.

[56] Yingying Xu et al. “FPGA Oriented Lightweight Deep Learning Inference for Liver Cancer Segmentation”. In: *2024 IEEE International Symposium on Biomedical Imaging (ISBI)*. IEEE, 2024, pp. 1–5. DOI: 10.1109/ISBI56570.2024.10635890.

[57] Zhanyuan Chang et al. “UNeXt: An Efficient Network for the Semantic Segmentation of High-Resolution Remote Sensing Images”. In: *Sensors* 24.20 (2024), p. 6655. DOI: 10.3390/s24206655.

[58] Walid Al-Dhabyani et al. “Dataset of Breast Ultrasound Images”. In: *Data in Brief* 28 (2020), p. 104863. DOI: 10.1016/j.dib.2019.104863.

[59] ISIC Challenge Organizers. *ISIC 2016: Skin Lesion Analysis Towards Melanoma Detection Challenge*. Accessed: 2025-09-22. 2016. URL: https://challenge.isic-archive.com/landing/2016/.

[60] Kai Jin et al. “FIVES: A Fundus Image Dataset for Artificial Intelligence Based Vessel Segmentation”. In: *Scientific Data* 9.1 (2022), p. 475. DOI: 10.1038/s41597-022-01564-3.

[61] Chunhui Chen et al. “Retinal Vessel Segmentation Using Deep Learning: A Review”. In: *IEEE Access* 9 (2021), pp. 111985–112004. DOI: 10.1109/ACCESS.2021.3102176.

[62] Andrew Howard et al. “Searching for MobileNetV3”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, 2019, pp. 1314–1324. DOI: 10.1109/ICCV.2019.00140.

[63] Jiacheng Ruan, Jincheng Li, and Suncheng Xiang. “VM-UNet: Vision Mamba UNet for Medical Image Segmentation”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications* (2024). DOI: 10.1145/3767748.

[64] Mingxing Tan and Quoc Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”. In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114. URL: https://proceedings.mlr.press/v97/tan19a.html.

[65] Sachin Mehta and Mohammad Rastegari. “MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer”. In: *arXiv* (2021). arXiv: 2110.02178 [cs.CV]. URL: https://arxiv.org/abs/2110.02178.

[66] Xilinx Inc. *Brevitas: Efficient Training and Quantization of Neural Networks*. Accessed: 2025-02-17. 2025. URL: https://github.com/Xilinx/brevitas.

[67] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”. In: *arXiv* (2015). eprint: 1510.00149. URL: https://arxiv.org/abs/1510.00149.

[68] Amir Gholami et al. “A Survey of Quantization Methods for Efficient Neural Network Inference”. In: *arXiv* (2021). eprint: 2103.13630. URL: https://arxiv.org/abs/2103.13630.

[69] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[70] Ron Banner, Yury Nahshan, and Daniel Soudry. “Post Training 4-Bit Quantization of Convolution Networks for Rapid-Deployment”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 32. 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/c1a26950ad46db23c27dfb80c9d80e4a-Abstract.html.

[71] Raghuraman Krishnamoorthi. “Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper”. In: *arXiv* (2018). eprint: 1806.08342. URL: https://arxiv.org/abs/1806.08342.

[72] Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. John Wiley & Sons, 2007. ISBN: 9780470127889.

[73] Yufei Ma et al. “Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, pp. 45–54. DOI: 10.1145/3020078.3021736.

[74] *Kria KV260 Vision AI Starter Kit*. AMD. 2025. URL: https://docs.amd.com/r/en-US/ds986-kv260-starter-kit (visited on 09/29/2025).

[75] *Ultra96-V2 | Avnet Boards*. Avnet. 2025. URL: https://www.avnet.com/americas/products/avnet-boards/avnet-board-families/ultra96-v2/ (visited on 09/29/2025).

[76] *Kria K26 SOM Data Sheet (DS987)*. Revision 1.5. AMD. 2024. URL: https://docs.amd.com/r/en-US/ds987-k26-som (visited on 09/29/2025).

[77] *Ultra96-V2 Hardware User's Guide, v1.3*. Avnet. 2020. URL: https://www.avnet.com/wps/wcm/connect/onesite/b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9/Ultra96-V2-HW-User-Guide-v1_3.pdf (visited on 09/29/2025).

A Python Operational Code

A.1 Training Code

```python
from collections import OrderedDict

from tqdm import tqdm

from utils.metrics import iou_score
from utils.util import AverageMeter


def train_network(
    model,
    trainloader,
    optimizer,
    criterion,
    scheduler,
    device,
    scheduler_type="batch",
    verbose=True,
):
    model.train()
    avg_meters = {
        "loss": AverageMeter(),
        "iou": AverageMeter(),
    }

    pbar = tqdm(total=len(trainloader)) if verbose else None

    for sampled_batch in trainloader:
        volume_batch, label_batch = sampled_batch["image"], sampled_batch["label"]
        volume_batch, label_batch = volume_batch.to(device), label_batch.to(device)

        outputs = model(volume_batch)
        loss = criterion(outputs, label_batch)
        iou, *_ = iou_score(outputs, label_batch)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        avg_meters["loss"].update(loss.item(), volume_batch.size(0))
        avg_meters["iou"].update(iou, volume_batch.size(0))

        # Scheduler update based on the type
        if scheduler_type == "batch":
            scheduler.step()

        if verbose:
```

```
            postfix = OrderedDict(
                [
                    ("loss", avg_meters["loss"].avg),
                    ("iou", avg_meters["iou"].avg),
                ]
            )
            pbar.set_postfix(postfix)
            pbar.update(1)

    if verbose:
        pbar.close()

    # If scheduler is epoch-based, call it here (e.g., ReduceLROnPlateau)
    if scheduler_type == "epoch":
        scheduler.step(avg_meters["loss"].avg)

    return OrderedDict([("loss", avg_meters["loss"].avg), ("iou", avg_meters["iou"].avg)])
```

Listing A.1: Python Train Code

A.2 Validation Code

```
from collections import OrderedDict

import torch
from tqdm import tqdm

from utils.metrics import iou_score
from utils.util import AverageMeter


def validate_network(model, valloader, criterion, device, verbose=True):
    avg_meters = {
        "val_loss": AverageMeter(),
        "val_iou": AverageMeter(),
        "val_dice": AverageMeter(),
        "SE": AverageMeter(),
        "PC": AverageMeter(),
        "F1": AverageMeter(),
        "AAC": AverageMeter(),
    }

    model.eval()

    with torch.no_grad():
        pbar = tqdm(total=len(valloader)) if verbose else None
        for sampled_batch in valloader:
            input, target = sampled_batch["image"], sampled_batch["label"]
            input = input.to(device)
            target = target.to(device)
            output = model(input)
            loss = criterion(output, target)

            iou, dice, SE, PC, F1, _, ACC = iou_score(output, target)
            avg_meters["val_loss"].update(loss.item(), input.size(0))
            avg_meters["val_iou"].update(iou, input.size(0))
            avg_meters["val_dice"].update(dice, input.size(0))
            avg_meters["SE"].update(SE, input.size(0))
            avg_meters["PC"].update(PC, input.size(0))
            avg_meters["F1"].update(F1, input.size(0))
            avg_meters["AAC"].update(ACC, input.size(0))
```

```
            if verbose:
                postfix = OrderedDict(
                    [
                        ("val_loss", avg_meters["val_loss"].avg),
                        ("val_iou", avg_meters["val_iou"].avg),
                        ("val_dice", avg_meters["val_dice"].avg),
                        ("SE", avg_meters["SE"].avg),
                        ("PC", avg_meters["PC"].avg),
                        ("F1", avg_meters["F1"].avg),
                        ("AAC", avg_meters["AAC"].avg),
                    ]
                )
                pbar.set_postfix(postfix)
                pbar.update(1)
        if verbose:
            pbar.close()

    return OrderedDict(
        [
            ("val_loss", avg_meters["val_loss"].avg),
            ("val_iou", avg_meters["val_iou"].avg),
            ("val_dice", avg_meters["val_dice"].avg),
            ("SE", avg_meters["SE"].avg),
            ("PC", avg_meters["PC"].avg),
            ("F1", avg_meters["F1"].avg),
            ("AAC", avg_meters["AAC"].avg),
        ]
    )
```

Listing A.2: Python Validation Code

A.3 Main Code

```
import argparse
from collections import OrderedDict
from datetime import datetime
import os
import random

import numpy as np
import pandas as pd
from thop import clever_format
import torch
import yaml

from dataloader.dataloader import get_dataloaders
from network import networks
from optimizers import optimizers, schedulers
from profilers.params import macs_and_params
from train import train_network
from utils import losses
from utils.util import str2bool
from validation import validate_network


dir_path = os.path.dirname(os.path.realpath(__file__))


def seed_torch(seed):
    np.random.seed(seed)
```

```
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
    random.seed(seed)
    np.random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)


def parse_args():
    # ...
    return vars(parser.parse_args())

# ...

def main():
    config = parse_args()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    setup(config)
    profile_network(config, device)

    model = networks.get_model(config["model"], config).to(device)
    trainloader, valloader = get_dataloaders(config)

    log = OrderedDict(
        [
            ("epoch", []),
            ("lr", []),
            ("loss", []),
            ("iou", []),
            ("val_loss", []),
            ("val_iou", []),
            ("val_dice", []),
            ("SE", []),
            ("PC", []),
            ("F1", []),
            ("AAC", []),
            ("best", []),
        ]
    )

    optimizer = optimizers.get_optimizer(config["optimizer"], model_params=model.parameters()
        , **config)
    criterion = losses.__dict__["BCEDiceLoss"]().to(device)
    scheduler = schedulers.get_scheduler(scheduler_name=config["scheduler"], optmzr=optimizer
        , **config)

    best_iou = 0
    trigger = 0

    for epoch in range(config["epochs"]):
        print("Epoch [%d/%d]" % (epoch, config["epochs"]))
        train_log = train_network(
            model,
            trainloader,
            optimizer,
            criterion,
            scheduler,
            device,
            verbose=config["verbose"],
        )
```

```
        val_log = validate_network(model, valloader, criterion, device, verbose=config["
            verbose"])
        print_logs(epoch, config["epochs"], train_log, val_log)

        log["epoch"].append(epoch)
        log["lr"].append(config["lr"])
        log["loss"].append(train_log["loss"])
        log["iou"].append(train_log["iou"])
        log["val_loss"].append(val_log["val_loss"])
        log["val_iou"].append(val_log["val_iou"])
        log["val_dice"].append(val_log["val_dice"])
        log["SE"].append(val_log["SE"])
        log["PC"].append(val_log["PC"])
        log["F1"].append(val_log["F1"])
        log["AAC"].append(val_log["AAC"])

        trigger += 1

        if val_log["val_iou"] > best_iou:
            best_iou = val_log["val_iou"]
            log["best"].append(True)
            save_log = {
                "epoch": epoch,
                "loss": float(log["val_loss"][-1]),
                "iou": float(log["val_iou"][-1]),
                "dice": float(log["val_dice"][-1]),
                "PC": float(log["PC"][-1]),
                "F1": float(log["F1"][-1]),
                "AAC": float(log["AAC"][-1]),
            }
            save(model, config["model_dir"], save_log)
            trigger = 0
        else:
            log["best"].append(False)

        pd.DataFrame(log).to_csv(os.path.join(config["model_dir"], "log.csv"), index=False)

        if 0 <= config["early_stopping"] <= trigger:
            print("=> Early stopping")
            break

        if torch.cuda.is_available():
            torch.cuda.empty_cache()

    config["end_datetime"] = datetime.now().strftime("%Y-%m-%d_%Hh%Mm%Ss")
    with open(os.path.join(config["model_dir"], "config.yml"), "w") as f:
        yaml.dump(config, f, sort_keys=False)


if __name__ == "__main__":
    main()
```

Listing A.3: Python Main Code

A.4 Inference Code

```
import argparse
from glob import glob
import os

from albumentations import Compose, Normalize, Resize
```

```python
import cv2
import numpy as np
import torch
import yaml

from network import networks


def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--model_dir",
        type=str,
        help="Trained model directory (needs files: model.pth, config.yml)",
        required=True,
    )
    parser.add_argument("--input", type=str, help="Input file to test the model", required=
        True)

    parser.add_argument("--mask", type=str, help="Mask file to test the model")

    parser.add_argument("--output_dir", type=str, help="Output directory", default="./")
    parser.add_argument(
        "--recursive", type=bool, help="Recursively infere every model in model_dir", default
            =False
    )
    parser.add_argument(
        "--dataset",
        type=str,
        help="Dataset for recursive inference",
    )
    args = parser.parse_args()
    if args.recursive and not args.dataset:
        parser.error("--recursive argument requires --dataset")

    return vars(args)


def load_yaml(file_path) -> dict:
    with open(file_path) as file:
        return yaml.safe_load(file)


def preprocess_image(config, image_path):
    transform = Compose(
        [
            Resize(config["input_h"], config["input_w"]),
            Normalize(),
        ]
    )
    img = cv2.imread(image_path)  # Ensure the image is in RGB format

    img = transform(image=img)["image"]
    img = img.astype(np.float32) / 255.0
    img = img.transpose(2, 0, 1)
    img = torch.tensor(img).unsqueeze(0)  # [C, H, W] -> [1, C, H, W]
    return img


def preprocess_mask(config, mask_path):
    transform = Compose(
        [
            Resize(config["input_h"], config["input_w"]),
```

```
            Normalize(),
        ]
    )
    mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
    mask = transform(image=mask)["image"]
    mask = mask.astype(np.float32) / 255.0
    mask = torch.tensor(mask).unsqueeze(0).unsqueeze(0)  # [H, W] -> [1, 1, H, W]
    return mask


def foreground_accuracy(output, mask):
    # Apply sigmoid and binarize predictions
    output = torch.sigmoid(output).cpu().numpy()
    output = (output >= 0.5).astype(np.uint8)

    # Binarize ground truth mask
    mask = torch.sigmoid(mask).cpu().numpy()
    mask = (mask >= 0.5).astype(np.uint8)

    if output.shape != mask.shape:
        raise ValueError(f"Mismatched shapes: {output.shape} vs {mask.shape}")

    # Compute confusion matrix components
    tp = np.logical_and(output == 1, mask == 1).sum()
    fp = np.logical_and(output == 1, mask == 0).sum()
    fn = np.logical_and(output == 0, mask == 1).sum()

    denominator = tp + fp + fn
    if denominator == 0:
        return float("nan")  # Or 0.0, depending on your preference

    foreground_acc = tp / denominator
    return foreground_acc


def load_model(config, model_pth, device):
    """Load the trained model."""
    model = networks.get_model(model_name=config["model"], config=config).to(device)
    state_dict = torch.load(model_pth, map_location=device, weights_only=True)
    model.load_state_dict(state_dict=state_dict, strict=False)
    return model.eval()


def save_output(config, output, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    output = torch.sigmoid(output).cpu().numpy()
    output[output >= 0.5] = 1
    output[output < 0.5] = 0
    output_name = (
        "_".join([config["dataset_name"], config["custom_name"] or config["model"]]) + config
            ["mask_ext"]
    )
    output_path = os.path.join(output_dir, output_name)

    for i in range(len(output)):
        for c in range(config["num_classes"]):
            cv2.imwrite(output_path, (output[i, c] * 255).astype("uint8"))


def infere_model(model_dir, input_path, mask_path, output_dir, device):
    config = load_yaml(os.path.join(model_dir, "config.yml"))
    model_pth = os.path.join(model_dir, "model.pth")
    model = load_model(config, model_pth, device)
```

```
    input_tensor = preprocess_image(config, input_path).to(device)

    with torch.inference_mode():
        output = model(input_tensor)

    if mask_path:
        mask_tensor = preprocess_mask(config, mask_path).to(device)
        acc = foreground_accuracy(output, mask_tensor)
        print(f"Foreground ACC: {acc}")

    save_output(config, output=output, output_dir=output_dir)
    print(f"Segmentation mask saved to {output_dir}")


def recursive_infere(data_dir, dataset, input_path, mask_path, output_dir, device):
    for model_path in glob(f"{data_dir}/*/"):
        dataset_dir = os.path.join(model_path, dataset)
        if os.path.exists(dataset_dir):
            timestamp_dirs = sorted(glob(f"{dataset_dir}/*/"), reverse=True)
            if timestamp_dirs:
                ts_dir = timestamp_dirs[0]
                config_file = os.path.join(ts_dir, "config.yml")
                model_file = os.path.join(ts_dir, "model.pth")

                if os.path.exists(config_file) and os.path.exists(model_file):
                    infere_model(ts_dir, input_path, mask_path, output_dir, device)


def infere(data_dir, input_path, dataset, mask_path, output_dir, device, recursive=False):
    if recursive:
        recursive_infere(data_dir, dataset, input_path, mask_path, output_dir, device)
    else:
        infere_model(data_dir, input_path, mask_path, output_dir, device)


def main():
    args = parse_args()
    model_dir = args["model_dir"]
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    infere(
        model_dir,
        args["input"],
        args["dataset"],
        args["mask"],
        args["output_dir"],
        device,
        recursive=args["recursive"],
    )


if __name__ == "__main__":
    main()
```

Listing A.4: Python Inference Code

A.5 Dataloader Code

```
import os

from albumentations import Compose, Flip, Normalize, RandomRotate90, Resize
```

```python
from torch.utils.data import DataLoader

from dataloader.dataset.DatasetSplit import DatasetSplit
from dataloader.dataset.MedicalDatasets import MedicalDataSets
from dataloader.dataset.MedicalDatasetsSplit import MedicalDataSetsSplit


datasets = {
    "ISIC2016": MedicalDataSetsSplit,
    "ISIC2017": MedicalDataSets,
    "FIVES2022": MedicalDataSetsSplit,
}


def get_transform(config, dataset_split=DatasetSplit.TRAIN):
    if dataset_split is DatasetSplit.TRAIN:
        return Compose(
            [
                RandomRotate90(),
                Flip(),
                Resize(config["input_h"], config["input_w"]),
                Normalize(),  # img = (img - mean * max_pixel_value) / (std * max_pixel_value)
            ]
        )
    return Compose(
        [
            Resize(config["input_h"], config["input_w"]),
            Normalize(),
        ]
    )


def get_dataloaders(config):
    dataset_root_train = os.path.join(config["data_dir"], config["dataset_name"], "train")
    if not (os.path.exists(dataset_root_train)) and len(os.listdir(dataset_root_train)) == 0:
        raise ValueError("No train dataset folder/data found!")

    dataset_class = datasets.get(config["dataset_name"], MedicalDataSetsSplit)

    if dataset_class == MedicalDataSetsSplit:
        MedicalDataSetsSplit.dataset_split(
            config["data_dir"], config["dataset_name"], config["img_ext"], val_size=0.3
        )

    db_train = dataset_class(
        data_dir=config["data_dir"],
        dataset_name=config["dataset_name"],
        transform=get_transform(config, dataset_split=DatasetSplit.TRAIN),
        img_ext=config["img_ext"],
        mask_ext=config["mask_ext"],
        num_classes=config["num_classes"],
        split=DatasetSplit.TRAIN,
    )

    trainloader = DataLoader(
        db_train,
        batch_size=config["batch_size"],
        shuffle=True,
        num_workers=config["num_workers"],
        pin_memory=True,
    )

```

```
    db_val = dataset_class(
        data_dir=config["data_dir"],
        dataset_name=config["dataset_name"],
        transform=get_transform(config, dataset_split=DatasetSplit.VAL),
        img_ext=config["img_ext"],
        mask_ext=config["mask_ext"],
        num_classes=config["num_classes"],
        split=DatasetSplit.VAL,
    )

    valloader = DataLoader(
        db_val,
        batch_size=config["batch_size"],
        shuffle=False,
        num_workers=config["num_workers"],
        pin_memory=True,
    )

    print(f"Train size: {len(db_train)}, Validation size: {len(db_val)}")

    return trainloader, valloader
```

Listing A.5: Python Dataloader Code

A.6 Medical Datasets Code

```
from glob import glob
import os

import cv2
import numpy as np
from torch.utils.data import Dataset

from dataloader.dataset.DatasetSplit import DatasetSplit


class MedicalDataSets(Dataset):
    def __init__(
        self,
        data_dir="./data",
        dataset_name="isic",
        img_ext=".png",
        mask_ext=".png",
        split: DatasetSplit = DatasetSplit.TRAIN,
        num_classes=1,
        transform=None,
    ):
        """
        Args:
            data_dir: Directory to data/ folder
            dataset_name: Name of the dataset (should match the folder name)
            img_ext (str): Image file extension.
            mask_ext (str): Mask file extension.
            num_classes (int): Number of classes.
            validation: Flag to indicate if it's a validation dataset.
            transform (Compose, optional): Albumentations transforms. Defaults to None.

        Folder structure:
            <dataset_name>
            |-- train
            |   |-- images
```

```
            |   `-- masks
            |       |-- 0
            |       |-- 1
            |       |-- ...
            |       `-- <num classes-1>
            |-- test (optional)
            |   |-- images
            |   `-- masks
            |       |-- 0
            |       |-- 1
            |       |-- ...
            |       `-- <num classes-1>
            `-- validation (optional)
                |-- images
                `-- masks
                    |-- 0
                    |-- 1
                    |-- ...
                    `-- <num classes-1>
        """
        self._base_dir = os.path.join(data_dir, dataset_name)
        self.img_dir = os.path.join(self._base_dir, split.value, "images")
        self.mask_dir = os.path.join(self._base_dir, split.value, "masks")
        self.img_ext = img_ext
        self.mask_ext = mask_ext
        self.num_classes = num_classes
        self.transform = transform

        # Verify folder structure and raise error if invalid
        if not os.path.exists(self.img_dir) or not os.path.exists(self.mask_dir):
            raise ValueError(f"Directory structure not found: {self.img_dir} or {self.
                mask_dir}")

        self.img_ids = [
            os.path.splitext(os.path.basename(p))[0]
            for p in glob(os.path.join(self.img_dir, "*" + self.img_ext))
        ]

    def __len__(self):
        return len(self.img_ids)

    def __getitem__(self, idx):
        case = self.img_ids[idx]

        img_path = os.path.join(self.img_dir, case + self.img_ext)
        img = cv2.imread(img_path)
        if img is None:
            raise FileNotFoundError(f"Image not found: {img_path}")

        mask = []
        for i in range(self.num_classes):
            mask_class_dir = os.path.join(self.mask_dir, str(i))
            mask_file = os.path.join(mask_class_dir, case + self.mask_ext)

            if os.path.isfile(mask_file):
                mask.append(cv2.imread(mask_file, cv2.IMREAD_GRAYSCALE)[..., None])
            else:
                matching_mask = glob(os.path.join(mask_class_dir, f"*{case}*{self.mask_ext}")
                    )
                if matching_mask:
                    mask.append(cv2.imread(matching_mask[0], cv2.IMREAD_GRAYSCALE)[..., None
                        ])
                else:
```

```
                    raise FileNotFoundError(f"Mask not found for case: {case} in {
                        mask_class_dir}")

        mask = np.dstack(mask)

        if self.transform is not None:
            augmented = self.transform(image=img, mask=mask)
            img = augmented["image"]
            mask = augmented["mask"]

        img = img.astype(np.float32) / 255.0
        img = img.transpose(2, 0, 1)
        mask = mask.astype(np.float32) / 255.0
        mask = mask.transpose(2, 0, 1)

        return {"image": img, "label": mask, "case": case}
```

Listing A.6: Python Medical Datasets Code

B PyTorch Networks Code

B.1 CMUNeXt PyTorch Code

```
import torch
from torch import nn


class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x):
        return self.fn(x) + x


class CMUNeXtBlock(nn.Module):
    def __init__(self, ch_in, ch_out, depth=1, k=3, act=nn.GELU):
        super().__init__()
        self.block = nn.Sequential(
            *[
                nn.Sequential(
                    Residual(
                        nn.Sequential(
                            # deep wise
                            nn.Conv2d(
                                ch_in,
                                ch_in,
                                kernel_size=(k, k),
                                groups=ch_in,
                                padding=(k // 2, k // 2),
                            ),
                            act(),
                            nn.BatchNorm2d(ch_in),
                        )
                    ),
                    nn.Conv2d(ch_in, ch_in * 4, kernel_size=(1, 1)),
                    act(),
                    nn.BatchNorm2d(ch_in * 4),
                    nn.Conv2d(ch_in * 4, ch_in, kernel_size=(1, 1)),
                    act(),
                    nn.BatchNorm2d(ch_in),
                )
                for _ in range(depth)
            ]
        )
        self.up = ConvBlock(ch_in, ch_out)

    def forward(self, x):
```

```
        x = self.block(x)
        x = self.up(x)
        return x


class ConvBlock(nn.Module):
    def __init__(self, ch_in, ch_out):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1, bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class UpConv(nn.Module):
    def __init__(self, ch_in, ch_out):
        super().__init__()
        self.up = nn.Sequential(
            nn.Upsample(scale_factor=2, mode="bilinear"),
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1, bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        x = self.up(x)
        return x


class FusionConv(nn.Module):
    def __init__(self, ch_in, ch_out, act=nn.GELU):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(ch_in, ch_in, kernel_size=3, stride=1, padding=1, groups=2, bias=True),
            act(),
            nn.BatchNorm2d(ch_in),
            nn.Conv2d(ch_in, ch_out * 4, kernel_size=(1, 1)),
            act(),
            nn.BatchNorm2d(ch_out * 4),
            nn.Conv2d(ch_out * 4, ch_out, kernel_size=(1, 1)),
            act(),
            nn.BatchNorm2d(ch_out),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class CMUNeXt(nn.Module):
    def __init__(
        self,
        input_channel=3,
        num_classes=1,
        dims=[16, 32, 128, 160, 256],
        depths=[1, 1, 1, 3, 1],
        kernels=[3, 3, 7, 7, 7],
        act=nn.GELU,
```

```
    ):
        """
        Args:
            input_channel : input channel.
            num_classes: output channel.
            dims: length of channels
            depths: length of cmunext blocks
            kernels: kernal size of cmunext blocks
        """
        super().__init__()
        # Encoder
        self.Maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.stem = ConvBlock(ch_in=input_channel, ch_out=dims[0])
        self.encoder1 = CMUNeXtBlock(ch_in=dims[0], ch_out=dims[0], depth=depths[0], k=
            kernels[0], act=act)
        self.encoder2 = CMUNeXtBlock(ch_in=dims[0], ch_out=dims[1], depth=depths[1], k=
            kernels[1], act=act)
        self.encoder3 = CMUNeXtBlock(ch_in=dims[1], ch_out=dims[2], depth=depths[2], k=
            kernels[2], act=act)
        self.encoder4 = CMUNeXtBlock(ch_in=dims[2], ch_out=dims[3], depth=depths[3], k=
            kernels[3], act=act)
        self.encoder5 = CMUNeXtBlock(ch_in=dims[3], ch_out=dims[4], depth=depths[4], k=
            kernels[4], act=act)
        # Decoder
        self.Up5 = UpConv(ch_in=dims[4], ch_out=dims[3])
        self.Up_conv5 = FusionConv(ch_in=dims[3] * 2, ch_out=dims[3], act=act)
        self.Up4 = UpConv(ch_in=dims[3], ch_out=dims[2])
        self.Up_conv4 = FusionConv(ch_in=dims[2] * 2, ch_out=dims[2], act=act)
        self.Up3 = UpConv(ch_in=dims[2], ch_out=dims[1])
        self.Up_conv3 = FusionConv(ch_in=dims[1] * 2, ch_out=dims[1], act=act)
        self.Up2 = UpConv(ch_in=dims[1], ch_out=dims[0])
        self.Up_conv2 = FusionConv(ch_in=dims[0] * 2, ch_out=dims[0], act=act)
        self.Conv_1x1 = nn.Conv2d(dims[0], num_classes, kernel_size=1)

    def forward(self, x):
        x1 = self.stem(x)
        x1 = self.encoder1(x1)
        x2 = self.Maxpool(x1)
        x2 = self.encoder2(x2)
        x3 = self.Maxpool(x2)
        x3 = self.encoder3(x3)
        x4 = self.Maxpool(x3)
        x4 = self.encoder4(x4)
        x5 = self.Maxpool(x4)
        x5 = self.encoder5(x5)

        d5 = self.Up5(x5)
        d5 = torch.cat((x4, d5), dim=1)
        d5 = self.Up_conv5(d5)

        d4 = self.Up4(d5)
        d4 = torch.cat((x3, d4), dim=1)
        d4 = self.Up_conv4(d4)

        d3 = self.Up3(d4)
        d3 = torch.cat((x2, d3), dim=1)
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        d2 = torch.cat((x1, d2), dim=1)
        d2 = self.Up_conv2(d2)
        d1 = self.Conv_1x1(d2)
        return d1
```

Listing B.1: CMUNeXt PyTorch Code

B.2 Mobile-CMUNeXt PyTorch Code

```
from torch import nn


class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x):
        return self.fn(x) + x


class MobileCMUNeXtBlock(nn.Module):
    def __init__(self, ch_in, ch_out, depth=1, k=3, act: nn.Module = nn.Hardswish):
        super().__init__()
        self.block = nn.Sequential(
            *[
                nn.Sequential(
                    # First depthwise convolution
                    Residual(
                        nn.Sequential(
                            nn.Conv2d(
                                ch_in,
                                ch_in,
                                kernel_size=(k, k),
                                groups=ch_in,
                                padding=(k // 2, k // 2),
                            ),
                            nn.BatchNorm2d(ch_in),
                            act(),
                        )
                    ),
                    # Second depthwise convolution
                    Residual(
                        nn.Sequential(
                            nn.Conv2d(
                                ch_in,
                                ch_in,
                                kernel_size=(k, k),
                                groups=ch_in,
                                padding=(k // 2, k // 2),
                            ),
                            nn.BatchNorm2d(ch_in),
                            act(),
                        )
                    ),
                    nn.Conv2d(ch_in, ch_in * 4, kernel_size=(1, 1)),
                    nn.BatchNorm2d(ch_in * 4),
                    act(),
                    nn.Conv2d(ch_in * 4, ch_in, kernel_size=(1, 1)),
                    nn.BatchNorm2d(ch_in),
                    act(),
                )
                for _ in range(depth)
            ]
```

```
        )
        self.up = ConvBlock(ch_in, ch_out, act=act)

    def forward(self, x):
        x = self.block(x)
        x = self.up(x)
        return x


class ConvBlock(nn.Module):
    def __init__(self, ch_in, ch_out, act: nn.Module = nn.Hardswish):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1, bias=True),
            nn.BatchNorm2d(ch_out),
            act(inplace=True),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class UpConv(nn.Module):
    def __init__(self, ch_in, ch_out, up_mode, act: nn.Module = nn.Hardswish):
        super().__init__()
        self.up = nn.Sequential(
            nn.Upsample(scale_factor=2, mode=up_mode),
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1, bias=True),
            nn.BatchNorm2d(ch_out),
            act(inplace=True),
        )

    def forward(self, x):
        x = self.up(x)
        return x


class FusionConv(nn.Module):
    def __init__(self, ch_in, ch_out, act: nn.Module = nn.Hardswish):
        super().__init__()
        self.conv = nn.Sequential(
            # nn.Conv2d(ch_in, ch_in, kernel_size=3, stride=1, padding=1, groups=2, bias=True
                ),
            nn.Conv2d(ch_in, ch_in, kernel_size=3, stride=1, padding=1, groups=1, bias=True),
            nn.BatchNorm2d(ch_in),
            act(),
            nn.Conv2d(ch_in, ch_out * 4, kernel_size=(1, 1)),
            nn.BatchNorm2d(ch_out * 4),
            act(),
            nn.Conv2d(ch_out * 4, ch_out, kernel_size=(1, 1)),
            nn.BatchNorm2d(ch_out),
            act(),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class MobileCMUNeXt_BN_ACT(nn.Module):
    def __init__(
        self,
```

```
        input_channel=3,
        num_classes=1,
        dims=[16, 32, 128, 160, 256],
        depths=[1, 1, 1, 3, 1],
        kernels=[3, 3, 7, 7, 7],
        upsampling_mode="nearest",
        act: nn.Module = nn.Hardswish,
    ):
        """
        Args:
            input_channel : input channel.
            num_classes: output channel.
            dims: length of channels
            depths: length of cmunext blocks
            kernels: kernal size of cmunext blocks

        Improvements Done:
            * Altered Concat to sum and altered shapes
            * Changed Activation Function to Hardswish
            * Added a second depthwise convolution
        """
        super().__init__()
        # Encoder
        self.Maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.stem = ConvBlock(ch_in=input_channel, ch_out=dims[0], act=act)
        self.encoder1 = MobileCMUNeXtBlock(
            ch_in=dims[0], ch_out=dims[0], depth=depths[0], k=kernels[0], act=act
        )
        self.encoder2 = MobileCMUNeXtBlock(
            ch_in=dims[0], ch_out=dims[1], depth=depths[1], k=kernels[1], act=act
        )
        self.encoder3 = MobileCMUNeXtBlock(
            ch_in=dims[1], ch_out=dims[2], depth=depths[2], k=kernels[2], act=act
        )
        self.encoder4 = MobileCMUNeXtBlock(
            ch_in=dims[2], ch_out=dims[3], depth=depths[3], k=kernels[3], act=act
        )
        self.encoder5 = MobileCMUNeXtBlock(
            ch_in=dims[3], ch_out=dims[4], depth=depths[4], k=kernels[4], act=act
        )


        # Decoder
        self.Up5 = UpConv(ch_in=dims[4], ch_out=dims[3], up_mode=upsampling_mode, act=act)
        self.Up_conv5 = FusionConv(ch_in=dims[3], ch_out=dims[3], act=act)
        self.Up4 = UpConv(ch_in=dims[3], ch_out=dims[2], up_mode=upsampling_mode, act=act)
        self.Up_conv4 = FusionConv(ch_in=dims[2], ch_out=dims[2], act=act)
        self.Up3 = UpConv(ch_in=dims[2], ch_out=dims[1], up_mode=upsampling_mode, act=act)
        self.Up_conv3 = FusionConv(ch_in=dims[1], ch_out=dims[1], act=act)
        self.Up2 = UpConv(ch_in=dims[1], ch_out=dims[0], up_mode=upsampling_mode, act=act)
        self.Up_conv2 = FusionConv(ch_in=dims[0], ch_out=dims[0], act=act)
        self.Conv_1x1 = nn.Conv2d(dims[0], num_classes, kernel_size=1, stride=1, padding=0)

    def forward(self, x):
        x = self.stem(x)
        x1 = self.encoder1(x)
        x2 = self.Maxpool(x1)
        x2 = self.encoder2(x2)
        x3 = self.Maxpool(x2)
        x3 = self.encoder3(x3)
        x4 = self.Maxpool(x3)
        x4 = self.encoder4(x4)
        x5 = self.Maxpool(x4)
```

```
        x5 = self.encoder5(x5)

        d5 = self.Up5(x5)
        d5 = x4 + d5
        d5 = self.Up_conv5(d5)

        d4 = self.Up4(d5)
        d4 = x3 + d4
        d4 = self.Up_conv4(d4)

        d3 = self.Up3(d4)
        d3 = x2 + d3
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        d2 = x1 + d2
        d2 = self.Up_conv2(d2)
        d1 = self.Conv_1x1(d2)
        return d1
```

Listing B.2: Mobile-CMUNeXt PyTorch Code

C Quantization Code

C.1 Mobile-CMUNeXt Quantized PyTorch Code With Batch Normalization

```
import brevitas.nn as qnn
from torch import nn

from network.mobile_cmunext.quantized.quant_config import (
    ActQuant,
    BiasQuant,
    WeightQuant,
    get_act_bit_width,
    get_weight_bit_width,
    set_bit_widths,
)


class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn
        self.input_quant = qnn.QuantIdentity(
            act_quant=ActQuant, bit_width=get_act_bit_width(), return_quant_tensor=True
        )

    def forward(self, x):
        fn_x = self.fn(x)
        qfn_x = self.input_quant(fn_x)
        qx = self.input_quant(x)
        return qfn_x + qx


class MobileCMUNeXtBlock(nn.Module):
    def __init__(self, ch_in, ch_out, depth=1, k=3, qact=qnn.QuantReLU):
        super().__init__()
        self.block = nn.Sequential(
            *[
                nn.Sequential(
                    # First depthwise convolution
                    Residual(
                        nn.Sequential(
                            qnn.QuantConv2d(
                                in_channels=ch_in,
                                out_channels=ch_in,
                                kernel_size=k,
                                stride=1,
                                padding=(k // 2, k // 2),
                                groups=ch_in,
```

```
                                bias=True,
                                bias_quant=BiasQuant,
                                weight_quant=WeightQuant,
                                weight_bit_width=get_weight_bit_width(),
                                return_quant_tensor=True,
                            ),
                            nn.BatchNorm2d(ch_in),
                            qact(
                                act_quant=ActQuant,
                                bit_width=get_act_bit_width(),
                                return_quant_tensor=True,
                            ),
                        )
                    ),
                    # Second depthwise convolution
                    Residual(
                        nn.Sequential(
                            qnn.QuantConv2d(
                                in_channels=ch_in,
                                out_channels=ch_in,
                                kernel_size=k,
                                stride=1,
                                padding=(k // 2, k // 2),
                                groups=ch_in,
                                bias=True,
                                bias_quant=BiasQuant,
                                weight_quant=WeightQuant,
                                weight_bit_width=get_weight_bit_width(),
                                return_quant_tensor=True,
                            ),
                            nn.BatchNorm2d(ch_in),
                            qact(
                                act_quant=ActQuant,
                                bit_width=get_act_bit_width(),
                                return_quant_tensor=True,
                            ),
                        )
                    ),
                    qnn.QuantConv2d(
                        in_channels=ch_in,
                        out_channels=ch_in * 4,
                        kernel_size=1,
                        bias=True,
                        bias_quant=BiasQuant,
                        weight_quant=WeightQuant,
                        weight_bit_width=get_weight_bit_width(),
                        return_quant_tensor=True,
                    ),
                    nn.BatchNorm2d(ch_in * 4),
                    qact(
                        act_quant=ActQuant,
                        bit_width=get_act_bit_width(),
                        return_quant_tensor=True,
                    ),
                    qnn.QuantConv2d(
                        in_channels=ch_in * 4,
                        out_channels=ch_in,
                        kernel_size=1,
                        bias=True,
                        bias_quant=BiasQuant,
                        weight_quant=WeightQuant,
                        weight_bit_width=get_weight_bit_width(),
                        return_quant_tensor=True,
```

```
                    ),
                    nn.BatchNorm2d(ch_in),
                    qact(
                        act_quant=ActQuant,
                        bit_width=get_act_bit_width(),
                        return_quant_tensor=True,
                    ),
                )
                for _ in range(depth)
            ]
        )
        self.up = ConvBlock(ch_in, ch_out, qact=qact)

    def forward(self, x):
        x = self.block(x)
        x = self.up(x)
        return x


class ConvBlock(nn.Module):
    def __init__(self, ch_in, ch_out, qact=qnn.QuantReLU):
        super().__init__()
        self.conv = nn.Sequential(
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_out,
                kernel_size=3,
                stride=1,
                padding=1,
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            nn.BatchNorm2d(ch_out),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class UpConv(nn.Module):
    def __init__(self, ch_in, ch_out, qact=qnn.QuantReLU):
        super().__init__()
        self.up = nn.Sequential(
            qnn.QuantUpsamplingBilinear2d(scale_factor=2, return_quant_tensor=True),
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_out,
                kernel_size=3,
                stride=1,
                padding=1,
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
```

```
                return_quant_tensor=True,
            ),
            nn.BatchNorm2d(ch_out),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                inplace=True,
                return_quant_tensor=True,
            ),
        )

    def forward(self, x):
        x = self.up(x)
        return x


class FusionConv(nn.Module):
    def __init__(self, ch_in, ch_out, qact=qnn.QuantReLU):
        super().__init__()
        self.conv = nn.Sequential(
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_in,
                kernel_size=3,
                stride=1,
                padding=1,
                groups=1,
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            nn.BatchNorm2d(ch_in),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_out * 4,
                kernel_size=(1, 1),
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            nn.BatchNorm2d(ch_out * 4),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantConv2d(
                in_channels=ch_out * 4,
                out_channels=ch_out,
                kernel_size=(1, 1),
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
```

```
                return_quant_tensor=True,
            ),
            nn.BatchNorm2d(ch_out),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class MobileCMUNeXt_Quant_BN_ACT(nn.Module):
    def __init__(
        self,
        weight_bit_width,
        act_bit_width,
        bias_bit_width,
        input_channel=3,
        num_classes=1,
        dims=[16, 32, 128, 160, 256],
        depths=[1, 1, 1, 3, 1],
        kernels=[3, 3, 7, 7, 7],
        qact=qnn.QuantReLU,
    ):
        """
        Args:
            input_channel : input channel.
            num_classes: output channel.
            dims: length of channels
            depths: length of cmunext blocks
            kernels: kernal size of cmunext blocks

        Improvements Done:
            * Altered Concat to sum and altered shapes
            * Changed Activation Function to Hardswish
            * Added a second depthwise convolution
        """
        super().__init__()

        set_bit_widths(weight_bit_width, act_bit_width, bias_bit_width)

        self.input_quant = qnn.QuantIdentity(act_quant=ActQuant, bit_width=8,
            return_quant_tensor=True)
        self.Maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.stem = ConvBlock(ch_in=input_channel, ch_out=dims[0], qact=qact)
        self.encoder1 = MobileCMUNeXtBlock(
            ch_in=dims[0], ch_out=dims[0], depth=depths[0], k=kernels[0], qact=qact
        )
        self.encoder2 = MobileCMUNeXtBlock(
            ch_in=dims[0], ch_out=dims[1], depth=depths[1], k=kernels[1], qact=qact
        )
        self.encoder3 = MobileCMUNeXtBlock(
            ch_in=dims[1], ch_out=dims[2], depth=depths[2], k=kernels[2], qact=qact
        )
        self.encoder4 = MobileCMUNeXtBlock(
            ch_in=dims[2], ch_out=dims[3], depth=depths[3], k=kernels[3], qact=qact
        )
        self.encoder5 = MobileCMUNeXtBlock(
            ch_in=dims[3], ch_out=dims[4], depth=depths[4], k=kernels[4], qact=qact
```

```
        )
        # Decoder
        self.Up5 = UpConv(ch_in=dims[4], ch_out=dims[3], qact=qact)
        self.Up_conv5 = FusionConv(ch_in=dims[3], ch_out=dims[3], qact=qact)
        self.Up4 = UpConv(ch_in=dims[3], ch_out=dims[2], qact=qact)
        self.Up_conv4 = FusionConv(ch_in=dims[2], ch_out=dims[2], qact=qact)
        self.Up3 = UpConv(ch_in=dims[2], ch_out=dims[1], qact=qact)
        self.Up_conv3 = FusionConv(ch_in=dims[1], ch_out=dims[1], qact=qact)
        self.Up2 = UpConv(ch_in=dims[1], ch_out=dims[0], qact=qact)
        self.Up_conv2 = FusionConv(ch_in=dims[0], ch_out=dims[0], qact=qact)
        self.Conv_1x1 = qnn.QuantConv2d(
            in_channels=dims[0],
            out_channels=num_classes,
            kernel_size=1,
            padding=0,
            bias=True,
            bias_quant=BiasQuant,
            weight_quant=WeightQuant,
            weight_bit_width=get_weight_bit_width(),
            return_quant_tensor=True,
        )
        self.x_quant = qnn.QuantIdentity(
            act_quant=ActQuant, bit_width=get_act_bit_width(), return_quant_tensor=True
        )
        self.output_quant = qnn.QuantIdentity(act_quant=ActQuant, bit_width=8,
            return_quant_tensor=True)

    def forward(self, x):
        x1 = self.input_quant(x)
        x1 = self.stem(x1)
        x1 = self.encoder1(x1)
        x2 = self.Maxpool(x1)
        x2 = self.encoder2(x2)
        x3 = self.Maxpool(x2)
        x3 = self.encoder3(x3)
        x4 = self.Maxpool(x3)
        x4 = self.encoder4(x4)
        x5 = self.Maxpool(x4)
        x5 = self.encoder5(x5)

        d5 = self.Up5(x5)
        x4 = self.x_quant(x4)
        d5 = self.x_quant(d5)
        d5 = x4 + d5
        d5 = self.Up_conv5(d5)

        d4 = self.Up4(d5)
        x3 = self.x_quant(x3)
        d4 = self.x_quant(d4)
        d4 = x3 + d4
        d4 = self.Up_conv4(d4)

        d3 = self.Up3(d4)
        x2 = self.x_quant(x2)
        d3 = self.x_quant(d3)
        d3 = x2 + d3
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        x1 = self.x_quant(x1)
        d2 = self.x_quant(d2)
        d2 = x1 + d2
        d2 = self.Up_conv2(d2)
```

```
        d1 = self.Conv_1x1(d2)
        d1 = self.output_quant(d1)
        return d1.tensor
```

Listing C.1: Mobile-CMUNeXt Quantized PyTorch Code With Batch Normalization

C.2 Mobile-CMUNeXt Quantized PyTorch Code Without Batch Normalization

```
import brevitas.nn as qnn
from torch import nn

from network.mobile_cmunext.quantized.quant_config import (
    ActQuant,
    BiasQuant,
    WeightQuant,
    get_act_bit_width,
    get_weight_bit_width,
    set_bit_widths,
)


class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn
        self.input_quant = qnn.QuantIdentity(
            act_quant=ActQuant, bit_width=get_act_bit_width(), return_quant_tensor=True
        )

    def forward(self, x):
        fn_x = self.fn(x)
        qfn_x = self.input_quant(fn_x)
        qx = self.input_quant(x)
        return qfn_x + qx


class MobileCMUNeXtBlock(nn.Module):
    def __init__(self, ch_in, ch_out, depth=1, k=3, qact=qnn.QuantReLU):
        super().__init__()
        self.block = nn.Sequential(
            *[
                nn.Sequential(
                    # First depthwise convolution
                    Residual(
                        nn.Sequential(
                            qnn.QuantConv2d(
                                in_channels=ch_in,
                                out_channels=ch_in,
                                kernel_size=k,
                                stride=1,
                                padding=(k // 2, k // 2),
                                groups=ch_in,
                                bias=True,
                                bias_quant=BiasQuant,
                                weight_quant=WeightQuant,
                                weight_bit_width=get_weight_bit_width(),
                                return_quant_tensor=True,
                            ),
                            qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
```

```
                        qact(
                            act_quant=ActQuant,
                            bit_width=get_act_bit_width(),
                            return_quant_tensor=True,
                        ),
                    )
                ),
                # Second depthwise convolution
                Residual(
                    nn.Sequential(
                        qnn.QuantConv2d(
                            in_channels=ch_in,
                            out_channels=ch_in,
                            kernel_size=k,
                            stride=1,
                            padding=(k // 2, k // 2),
                            groups=ch_in,
                            bias=True,
                            bias_quant=BiasQuant,
                            weight_quant=WeightQuant,
                            weight_bit_width=get_weight_bit_width(),
                            return_quant_tensor=True,
                        ),
                        qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
                        qact(
                            act_quant=ActQuant,
                            bit_width=get_act_bit_width(),
                            return_quant_tensor=True,
                        ),
                    )
                ),
                qnn.QuantConv2d(
                    in_channels=ch_in,
                    out_channels=ch_in * 4,
                    kernel_size=1,
                    bias=True,
                    bias_quant=BiasQuant,
                    weight_quant=WeightQuant,
                    weight_bit_width=get_weight_bit_width(),
                    return_quant_tensor=True,
                ),
                qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
                qact(
                    act_quant=ActQuant,
                    bit_width=get_act_bit_width(),
                    return_quant_tensor=True,
                ),
                qnn.QuantConv2d(
                    in_channels=ch_in * 4,
                    out_channels=ch_in,
                    kernel_size=1,
                    bias=True,
                    bias_quant=BiasQuant,
                    weight_quant=WeightQuant,
                    weight_bit_width=get_weight_bit_width(),
                    return_quant_tensor=True,
                ),
                qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
                qact(
                    act_quant=ActQuant,
                    bit_width=get_act_bit_width(),
                    return_quant_tensor=True,
                ),
```

```
                )
                for _ in range(depth)
            ]
        )
        self.up = ConvBlock(ch_in, ch_out, qact=qact)

    def forward(self, x):
        x = self.block(x)
        x = self.up(x)
        return x


class ConvBlock(nn.Module):
    def __init__(self, ch_in, ch_out, qact=qnn.QuantReLU):
        super().__init__()
        self.conv = nn.Sequential(
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_out,
                kernel_size=3,
                stride=1,
                padding=1,
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class UpConv(nn.Module):
    def __init__(self, ch_in, ch_out, qact=qnn.QuantReLU):
        super().__init__()
        self.up = nn.Sequential(
            qnn.QuantUpsamplingBilinear2d(scale_factor=2, return_quant_tensor=True),
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_out,
                kernel_size=3,
                stride=1,
                padding=1,
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                inplace=True,
```

```
                return_quant_tensor=True,
            ),
        )

    def forward(self, x):
        x = self.up(x)
        return x


class FusionConv(nn.Module):
    def __init__(self, ch_in, ch_out, qact=qnn.QuantReLU):
        super().__init__()
        self.conv = nn.Sequential(
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_in,
                kernel_size=3,
                stride=1,
                padding=1,
                groups=1,
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantConv2d(
                in_channels=ch_in,
                out_channels=ch_out * 4,
                kernel_size=(1, 1),
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantConv2d(
                in_channels=ch_out * 4,
                out_channels=ch_out,
                kernel_size=(1, 1),
                bias=True,
                bias_quant=BiasQuant,
                weight_quant=WeightQuant,
                weight_bit_width=get_weight_bit_width(),
                return_quant_tensor=True,
            ),
            qnn.QuantIdentity(act_quant=None, return_quant_tensor=True),
            qact(
                act_quant=ActQuant,
                bit_width=get_act_bit_width(),
                return_quant_tensor=True,
```

```
            ),
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class MobileCMUNeXt_Quant_ACT(nn.Module):
    def __init__(
        self,
        weight_bit_width,
        act_bit_width,
        bias_bit_width,
        input_channel=3,
        num_classes=1,
        dims=[16, 32, 128, 160, 256],
        depths=[1, 1, 1, 3, 1],
        kernels=[3, 3, 7, 7, 7],
        qact=qnn.QuantReLU,
    ):
        """
        Args:
            input_channel : input channel.
            num_classes: output channel.
            dims: length of channels
            depths: length of cmunext blocks
            kernels: kernal size of cmunext blocks

        Improvements Done:
            * Altered Concat to sum and altered shapes
            * Changed Activation Function to Hardswish
            * Added a second depthwise convolution
        """
        super().__init__()

        set_bit_widths(weight_bit_width, act_bit_width, bias_bit_width)

        self.input_quant = qnn.QuantIdentity(act_quant=ActQuant, bit_width=8,
            return_quant_tensor=True)
        self.Maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.stem = ConvBlock(ch_in=input_channel, ch_out=dims[0], qact=qact)
        self.encoder1 = MobileCMUNeXtBlock(
            ch_in=dims[0], ch_out=dims[0], depth=depths[0], k=kernels[0], qact=qact
        )
        self.encoder2 = MobileCMUNeXtBlock(
            ch_in=dims[0], ch_out=dims[1], depth=depths[1], k=kernels[1], qact=qact
        )
        self.encoder3 = MobileCMUNeXtBlock(
            ch_in=dims[1], ch_out=dims[2], depth=depths[2], k=kernels[2], qact=qact
        )
        self.encoder4 = MobileCMUNeXtBlock(
            ch_in=dims[2], ch_out=dims[3], depth=depths[3], k=kernels[3], qact=qact
        )
        self.encoder5 = MobileCMUNeXtBlock(
            ch_in=dims[3], ch_out=dims[4], depth=depths[4], k=kernels[4], qact=qact
        )
        # Decoder
        self.Up5 = UpConv(ch_in=dims[4], ch_out=dims[3], qact=qact)
        self.Up_conv5 = FusionConv(ch_in=dims[3], ch_out=dims[3], qact=qact)
        self.Up4 = UpConv(ch_in=dims[3], ch_out=dims[2], qact=qact)
        self.Up_conv4 = FusionConv(ch_in=dims[2], ch_out=dims[2], qact=qact)
        self.Up3 = UpConv(ch_in=dims[2], ch_out=dims[1], qact=qact)
```

```
        self.Up_conv3 = FusionConv(ch_in=dims[1], ch_out=dims[1], qact=qact)
        self.Up2 = UpConv(ch_in=dims[1], ch_out=dims[0], qact=qact)
        self.Up_conv2 = FusionConv(ch_in=dims[0], ch_out=dims[0], qact=qact)
        self.Conv_1x1 = qnn.QuantConv2d(
            in_channels=dims[0],
            out_channels=num_classes,
            kernel_size=1,
            padding=0,
            bias=True,
            bias_quant=BiasQuant,
            weight_quant=WeightQuant,
            weight_bit_width=get_weight_bit_width(),
            return_quant_tensor=True,
        )
        self.x_quant = qnn.QuantIdentity(
            act_quant=ActQuant, bit_width=get_act_bit_width(), return_quant_tensor=True
        )
        self.output_quant = qnn.QuantIdentity(act_quant=ActQuant, bit_width=8,
            return_quant_tensor=True)

    def forward(self, x):
        x1 = self.input_quant(x)
        x1 = self.stem(x1)
        x1 = self.encoder1(x1)
        x2 = self.Maxpool(x1)
        x2 = self.encoder2(x2)
        x3 = self.Maxpool(x2)
        x3 = self.encoder3(x3)
        x4 = self.Maxpool(x3)
        x4 = self.encoder4(x4)
        x5 = self.Maxpool(x4)
        x5 = self.encoder5(x5)

        d5 = self.Up5(x5) # uspample (C3D) output scale OK
        x4 = self.x_quant(x4)
        d5 = self.x_quant(d5)
        d5 = x4 + d5
        d5 = self.Up_conv5(d5) # fusion (C3D) skip con scale MODIFY

        d4 = self.Up4(d5)
        x3 = self.x_quant(x3)
        d4 = self.x_quant(d4)
        d4 = x3 + d4
        d4 = self.Up_conv4(d4)

        d3 = self.Up3(d4)
        x2 = self.x_quant(x2)
        d3 = self.x_quant(d3)
        d3 = x2 + d3
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        x1 = self.x_quant(x1)
        d2 = self.x_quant(d2)
        d2 = x1 + d2
        d2 = self.Up_conv2(d2)
        d1 = self.Conv_1x1(d2)
        d1 = self.output_quant(d1)
        return d1.tensor
```

Listing C.2: Mobile-CMUNeXt Quantized PyTorch Code Without Batch Normalization

C.3 Mobile-CMUNeXt Quantization Configuration PyTorch Code

```python
from brevitas.core.bit_width import BitWidthImplType
from brevitas.core.quant import QuantType
from brevitas.core.restrict_val import FloatToIntImplType, RestrictValueType
from brevitas.core.scaling import ScalingImplType
from brevitas.core.zero_point import ZeroZeroPoint
from brevitas.inject import ExtendedInjector
from brevitas.inject.enum import StatsOp
from brevitas.quant import IntBias
from brevitas.quant.solver import ActQuantSolver, WeightQuantSolver
from dependencies import value


WEIGHT_BIT_WIDTH = 8
ACT_BIT_WIDTH = 8  # more sensitive
BIAS_BIT_WIDTH = 16  # no need to alter


def set_bit_widths(
    weight_bit_width=8,
    act_bit_width=8,
    bias_bit_width=16,
):
    global WEIGHT_BIT_WIDTH  # noqa: PLW0603
    global ACT_BIT_WIDTH  # noqa: PLW0603
    global BIAS_BIT_WIDTH  # noqa: PLW0603

    WEIGHT_BIT_WIDTH = weight_bit_width
    ACT_BIT_WIDTH = act_bit_width
    BIAS_BIT_WIDTH = bias_bit_width


def get_weight_bit_width():
    global WEIGHT_BIT_WIDTH  # noqa: PLW0602
    return WEIGHT_BIT_WIDTH


def get_act_bit_width():
    global ACT_BIT_WIDTH  # noqa: PLW0602
    return ACT_BIT_WIDTH


class BaseQuant(ExtendedInjector):
    """
    A base quantization configuration class that defines common properties
    for different quantization types (weights, activations, biases).
    """

    bit_width_impl_type = BitWidthImplType.CONST
    restrict_scaling_type = RestrictValueType.POWER_OF_TWO
    zero_point_impl = ZeroZeroPoint
    float_to_int_impl_type = FloatToIntImplType.ROUND
    scaling_impl_type = ScalingImplType.STATS
    scaling_stats_op = StatsOp.MAX
    scaling_per_output_channel = False
    bit_width = None
    signed = True
    narrow_range = True

    @value
    def quant_type():
```

```
        """
        Determines the quantization type based on the weight bit-width.
        """
        return QuantType.BINARY if WEIGHT_BIT_WIDTH == 1 else QuantType.INT


class WeightQuant(BaseQuant, WeightQuantSolver):
    """
    Quantization configuration for weights.
    """

    scaling_const = 1.0


class ActQuant(BaseQuant, ActQuantSolver):
    """
    Quantization configuration for activations.
    """

    signed = True


class BiasQuant(BaseQuant, IntBias):
    """
    Quantization configuration for biases.
    """

    bit_width = BIAS_BIT_WIDTH
```

Listing C.3: Mobile-CMUNeXt Quantization Configuration PyTorch Code

C.4 Quantized Weights Extraction Code

```
from abc import ABC, abstractmethod
import argparse
import os

from brevitas.nn import QuantConv2d
import numpy as np
import torch
import yaml

from network import networks


def float_to_fixed_array(values, scale, dtype=np.int8):
    values = np.asarray(values, dtype=np.float32)
    scale = np.asarray(scale, dtype=np.float32)
    q = np.round(values / scale)
    return q.astype(dtype)


def bit_width_to_dtype(bit_width, signed=True):
    """
    Returns a NumPy integer dtype corresponding to the given bit width.
    Parameters:
        bit_width (int): Bit width of the desired integer type (8, 16, 32, 64).
        signed (bool): Whether to use signed integers. Default is True.
    Returns:
        np.dtype: Corresponding NumPy dtype (e.g., np.int8, np.uint16).
    Raises:
```

```
        ValueError: If the bit width is not supported.
    """
    dtype_map = {
        8:  (np.int8,  np.uint8),
        16: (np.int16, np.uint16),
        32: (np.int32, np.uint32),
        64: (np.int64, np.uint64),
    }

    if bit_width not in dtype_map:
        raise ValueError(f"Unsupported bit width: {bit_width}. Must be one of: {list(dtype_map.keys())}")
    return dtype_map[bit_width][0] if signed else dtype_map[bit_width][1]



def load_yaml(file_path) -> dict:
    with open(file_path) as file:
        return yaml.safe_load(file)


def setcache(m):
    m.cache_inference_quant_bias = True


def parse_args():
    parser = argparse.ArgumentParser()

    # base
    parser.add_argument(
        "--model_dir",
        type=str,
        help="Trained model directory (needs files: model.pth, config.yml)",
        required=True,
    )
    return vars(parser.parse_args())


def load_model(config, model_pth, device):
    """
    Load the trained model.
    """
    model = networks.get_model(model_name=config["model"], config=config).to(device)
    model.apply(setcache)
    state_dict = torch.load(model_pth, map_location=device, weights_only=True)
    model.load_state_dict(state_dict=state_dict, strict=False)
    return model.eval()


def generate_input(config, device):
    C = config["input_channels"]
    H = config["input_h"]
    W = config["input_w"]
    return torch.zeros(1, C, H, W).to(device)


def array_2_string_cstyle(arr, idx, indent=4):
    """
    Convert a NumPy array to a nicely formatted C-style initializer (just the body, no declaration),
    with a C comment indicating the shape.
    """
    def format_array(a, level=1):
```

```
        spacing = ' ' * (indent * level)
        if a.ndim == 1:
            elements = ', '.join(f"{x:2}" for x in a)
            return spacing + "{ " + elements + "}"

        inner = ',\n'.join(format_array(sub, level+1) for sub in a)
        return spacing + "{\n" + inner + "\n" + spacing + "}"

    c_comment = f"/* idx: {idx}, shape: {arr.shape} */\n"
    return c_comment + format_array(arr, 0) + ","


def delete_if_found(file_path):
    if os.path.exists(file_path):
        os.remove(file_path)



class ParametersCppBase(ABC):
    def __init__(self, output_dir, weight_filename, bias_filename):
        self.weight_name = os.path.splitext(weight_filename)[0]
        self.bias_name = os.path.splitext(bias_filename)[0]
        os.makedirs(output_dir, exist_ok=True)

        self.weigths_file = os.path.join(output_dir, weight_filename)
        if os.path.exists(self.weigths_file):
            os.remove(self.weigths_file)

        self.bias_file = os.path.join(output_dir, bias_filename)
        if os.path.exists(self.bias_file):
            os.remove(self.bias_file)

        self.w_counter = 0
        self.b_counter = 0
        self.w_max_shape = None
        self.b_max_shape = None

    @abstractmethod
    def handle_weights(self, fixed_weights):
        """Transform the weights into the desired shape."""

    def _array_2_string_cstyle(self, arr, idx, indent=4):
        """
        Convert a NumPy array to a nicely formatted C-style initializer,
        with a C comment indicating index and shape.
        """
        def format_array(a, level=1):
            spacing = ' ' * (indent * level)
            if a.ndim == 1:
                elements = ', '.join(f"{x:2}" for x in a)
                return spacing + "{ " + elements + "}"

            inner = ',\n'.join(format_array(sub, level+1) for sub in a)
            return spacing + "{\n" + inner + "\n" + spacing + "}"

        c_comment = f"\n\n/* idx: {idx}, shape: {arr.shape} */\n"
        return c_comment + format_array(arr, 1) + ","

    def write_weights(self, fixed_weights):
        handled_weights = self.handle_weights(fixed_weights)
        c_string = self._array_2_string_cstyle(handled_weights, self.w_counter)
        with open(self.weigths_file, "a") as f:
            if self.w_counter == 0:
```

```
                self.w_max_shape = handled_weights.shape
                brackets = "[]" * handled_weights.ndim
                f.write("// Auto-generated from quantized weights model\n")
                f.write("// See end of map declaration to find max map size\n")
                f.write(f"const static int {self.weight_name}[]{brackets} = {{")
            else:
                self.w_max_shape = tuple(max(a, b) for a, b in zip(self.w_max_shape,
                    handled_weights.shape, strict=False))
            f.write(c_string)

        self.w_counter += 1

    def write_bias(self, fixed_bias):
        c_string = self._array_2_string_cstyle(fixed_bias, self.w_counter)
        with open(self.bias_file, "a") as f:
            if self.b_counter == 0:
                self.b_max_shape = fixed_bias.shape
                brackets = "[]" * fixed_bias.ndim
                f.write("// Auto-generated from quantized weights model\n")
                f.write("// See end of map declaration to find max map size\n")
                f.write(f"const static int {self.bias_name}[]{brackets} = {{")
            else:
                self.b_max_shape = tuple(max(a, b) for a, b in zip(self.b_max_shape,
                    fixed_bias.shape, strict=False))
            f.write(c_string)

        self.b_counter += 1

    def cleanup(self):
        if self.w_counter > 0:
            with open(self.weigths_file, "a") as f:
                f.write("\n};\n")
                dims_brackets = ''.join(f'[{d}]' for d in self.w_max_shape)
                f.write(f"// [{self.w_counter}]{dims_brackets} \n")

        if self.b_counter > 0:
            with open(self.bias_file, "a") as f:
                f.write("\n};\n")
                dims_brackets = ''.join(f'[{d}]' for d in self.b_max_shape)
                f.write(f"// [{self.w_counter}]{dims_brackets} \n")

class ParametersPWCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weightsPW.h", "biasPW.h")

    def handle_weights(self, fixed_weights):
        # (outC, inC, 1, 1) -> (outC, inC)
        return np.squeeze(fixed_weights, axis=(2, 3))


class ParametersDWCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weightsDW.h", "biasDW.h")

    def handle_weights(self, fixed_weights):
        # (outC, 1, kH, kW) -> (kH, kW, outC)
        return np.squeeze(fixed_weights, axis=1).transpose(1, 2, 0)

class Parameters3DCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weights3D.h", "bias3D.h")

    def handle_weights(self, fixed_weights):
```

```
        # (outC, inC, kH, kW) -> (kH, kW, outC, inC)
        return fixed_weights.transpose(2, 3, 0, 1)

# class ParametersG2Cpp(ParametersCppBase):
#     def __init__(self, output_dir):
#         super().__init__(output_dir, "weightsG2.h", "biasG2.h")

#     def handle_weights(self, fixed_weights):
#         # (outC, inC, kH, kW) -> (kH, kW, outC, inC)
#         return  fixed_weights.transpose(2, 3, 0, 1)



def extract_weights_and_bias(model, output_dir, device):
    os.makedirs(output_dir, exist_ok=True)
    id = 0

    weightsDWCpp = ParametersDWCpp(output_dir)
    weightsPWCpp = ParametersPWCpp(output_dir)
    # weightsG2Cpp = ParametersG2Cpp(output_dir)
    weights3DCpp = Parameters3DCpp(output_dir)
    weights_cpp_class : ParametersCppBase = None

    for name, module in model.named_modules():
        base_name = f"{id}_{name}_{module.__class__.__name__}"
        description_file_name = f"{base_name}.txt"
        debug_file_name =  f"{base_name}.values"
        bin_file_name = f"{base_name}.bin"
        description_file = os.path.join(output_dir, description_file_name)
        bin_file = os.path.join(output_dir, bin_file_name)
        debug_file = os.path.join(output_dir, debug_file_name)

        delete_if_found(description_file)
        delete_if_found(bin_file)
        delete_if_found(debug_file)

        if isinstance(module, QuantConv2d):

            if module.groups == module.in_channels and module.in_channels == module.
                out_channels:
                weights_cpp_class = weightsDWCpp
            elif module.kernel_size == (1, 1) and module.groups == 1:
                weights_cpp_class = weightsPWCpp
            # elif module.groups == 2:
                # weights_cpp_class = weightsG2Cpp
            else:
                weights_cpp_class = weights3DCpp


            # === Weights ===
            if hasattr(module, "weight") and hasattr(module, "weight_quant"):
                id+=1
                quant_tensor = module.quant_weight()
                scale = getattr(quant_tensor, "scale", None)
                scale = scale.item() if hasattr(scale, 'item') else scale
                signed = getattr(quant_tensor, "signed_t", None)
                weights = getattr(quant_tensor, "value", None)
                bit_width = getattr(quant_tensor, "bit_width", None)
                dtype = bit_width_to_dtype(int(bit_width), signed)

                weights_np = weights.detach().to(device).numpy()
                fixed_values = float_to_fixed_array(weights_np, scale, dtype)

```

```
                with open(description_file, "a") as f:
                    f.write("#" * 80 + "\n")
                    f.write(f"[Weight] Layer: {name} ({module.__class__.__name__})\n")
                    f.write(f"  Shape: {tuple(quant_tensor.shape)}\n")
                    f.write(f"  Bit Width: {bit_width.item()}\n")
                    f.write(f"  Signed: {signed}\n")
                    f.write(f"  Scale: {scale}\n")
                    f.write(
                        f"  Sample Weights: {np.array2string(weights_np[:10], precision=5)}...\n"
                    )
                    f.write("-" * 80 + "\n")

                with open(debug_file, "a") as f:
                    f.write(np.array2string(weights_np, precision=5))
                    f.write("\nFixed Point: \n" + np.array2string(fixed_values))

                with open(bin_file, "ab") as f:
                    f.write(fixed_values.tobytes())

                # ----------------------------------C FILE STORE ----------------------------------
                weights_cpp_class.write_weights(fixed_values)

            # === Bias ===
            if hasattr(module, "bias_quant"):
                quant_tensor = module.quant_bias()
                scale = getattr(quant_tensor, "scale", None)
                scale = scale.item() if hasattr(scale, 'item') else scale
                signed = getattr(quant_tensor, "signed_t", None)
                bias = getattr(quant_tensor, "value", None)
                bit_width = getattr(quant_tensor, "bit_width", None)

                bias_np = bias.detach().to(device).numpy()
                array_flat = bias_np.flatten(order="C")

                with open(description_file, "a") as f:
                    f.write(f"[Bias] Layer: {name} ({module.__class__.__name__})\n")
                    f.write(f"  Shape: {tuple(quant_tensor.shape)}\n")
                    f.write(f"  Bit Width: {bit_width.item()}\n")
                    f.write(f"  Signed: {signed}\n")
                    f.write(f"  Scale: {scale}\n")
                    f.write(
                        f"  Sample Bias: {np.array2string(array_flat[:10], precision=5)}...\n"
                    )
                    f.write("#" * 80 + "\n\n")

                dtype = bit_width_to_dtype(int(bit_width), signed)
                fixed_values = float_to_fixed_array(array_flat, scale, dtype)

                with open(debug_file, "a") as f:
                    f.write("\n" + "-" * 80 + "\n")
                    f.write(np.array2string(array_flat, precision=5))
                    f.write("\nFixed Point: \n" + np.array2string(fixed_values))

                with open(bin_file, "ab") as f:
                    f.write(fixed_values.tobytes())

                # ----------------------------------C FILE STORE ----------------------------------
                weights_cpp_class.write_bias(fixed_values)

```

```
335
336     # DONT FORGET
337     weightsDWCpp.cleanup()
338     weightsPWCpp.cleanup()
339     weights3DCpp.cleanup()
340     # weightsG2Cpp.cleanup()
341
342
343 def weight_extractor(device, model_dir):
344     config = load_yaml(os.path.join(model_dir, "config.yml"))
345     model_pth = os.path.join(model_dir, "model.pth")
346     model = load_model(config, model_pth, device)
347
348     output_dir = os.path.join(model_dir, "weights")
349
350     input_ = generate_input(config, device)
351     model(input_)  # Run a dummy inference first
352     extract_weights_and_bias(model, output_dir, device)
```

Listing C.4: Quantized Weights Extraction Code

C.5 Quantized Scale Export Code

```
1  #!/usr/bin/env python3
2  import argparse
3  from collections import deque
4  import math
5  import os
6
7  import brevitas.nn as qnn
8  import torch
9  from torch import nn
10 import yaml
11
12 from network import networks
13
14
15 # ---------------------------
16 # Boilerplate
17 # ---------------------------
18 def setcache(m):
19     m.cache_inference_quant_bias = True
20
21
22 def load_yaml(path) -> dict:
23     with open(path) as f:
24         return yaml.safe_load(f)
25
26
27 def parse_args():
28     p = argparse.ArgumentParser(
29         "Extract per-Residual CODE shifts to map {x, fn(x)} scales to the new input_quant
              scale"
30     )
31     p.add_argument("--model_dir", type=str, required=True,
32                    help="Trained model directory (needs files: model.pth, config.yml)")
33     return vars(p.parse_args())
34
35
36 def load_model(config, model_pth, device):
37     model = networks.get_model(model_name=config["model"], config=config).to(device)
```

```
    model.apply(setcache)
    state = torch.load(model_pth, map_location=device, weights_only=True)
    model.load_state_dict(state, strict=True)
    model.eval().to(device)
    return model


def generate_input(config, device):
    C, H, W = config["input_channels"], config["input_h"], config["input_w"]
    return torch.zeros(1, C, H, W, device=device)


def _scale_of(qt):
    """
    Extract numeric scale from a Brevitas QuantTensor or return None.
    Works for inputs (pre) and outputs (post) of QuantIdentity.
    """
    if qt is None:
        return None
    s = getattr(qt, "scale", None)
    if s is None:
        return None
    try:
        return float(s.item()) if hasattr(s, "item") else float(s)
    except Exception:
        return None


# --------------------------
# Header writer (2-D array)
# --------------------------
class Residual2DWriter:
    def __init__(self, out_dir, filename="scaleRES.h", array_name="scaleRES"):
        os.makedirs(out_dir, exist_ok=True)
        self.path = os.path.join(out_dir, filename)
        self.array_name = array_name
        if os.path.exists(self.path):
            os.remove(self.path)
        self.rows = 0
        self._opened = False

    def _open_if_needed(self):
        if self._opened:
            return
        with open(self.path, "a") as f:
            f.write("// Auto-generated from quantized model\n")
            f.write("// scaleRES[layerID][2] = { x_shift, fn_shift }\n")
            f.write("// Each entry is the CODE shift k to map s_in -> s_q:\n")
            f.write("//   k = round(log2(s_q / s_in))\n")
            f.write("//   k > 0 : code RIGHT shift by k  (>>)  [s_q > s_in]\n")
            f.write("//   k < 0 : code LEFT  shift by |k| (<<) [s_q < s_in]\n")
            f.write(f"static const int {self.array_name}[][2] = {{\n")
        self._opened = True

    def add_row(self, x_shift: int, fn_shift: int, comment: str = ""):
        self._open_if_needed()
        with open(self.path, "a") as f:
            if comment:
                f.write(f"  /* layer {self.rows}: {comment} */ ")
            else:
                f.write(f"  /* layer {self.rows} */ ")
            f.write(f"{{ {int(x_shift)}, {int(fn_shift)} }},\n")
        self.rows += 1
```

```
101
102     def close(self):
103         if not self._opened:
104             with open(self.path, "a") as f:
105                 f.write(f"static const int {self.array_name}[][2] = {{}};\n")
106                 f.write(f"static const int {self.array_name}_ROWS = 0;\n")
107             return
108         with open(self.path, "a") as f:
109             f.write("};\n")
110             f.write(f"static const int {self.array_name}_ROWS = {self.rows};\n")
111
112
113 # --------------------------
114 # Residual hooks
115 # --------------------------
116 def attach_residual_hooks(model: nn.Module, output_dir="./output"):
117     """
118     Targets modules that look like your Residual wrapper:
119       - attribute `input_quant` (QuantIdentity), called twice per forward
120       - attribute `fn` (Module)
121     For each `input_quant` call we record:
122         s_in  = scale of the *input* QuantTensor
123         s_q   = scale of the *output* QuantTensor
124     Then per Residual we compute CODE shifts:
125         k_x  = round(log2(s_qx  / s_x ))
126         k_fn = round(log2(s_qfn / s_fn))
127     and write a row { k_x, k_fn }.
128     """
129     writer = Residual2DWriter(output_dir)
130     name_of = {m: n for n, m in model.named_modules()}
131
132     buffers = {}
133     hooks = []
134
135     # Detect Residual modules structurally
136     residuals = []
137     for m in model.modules():
138         if (
139             hasattr(m, "input_quant")
140             and isinstance(m.input_quant, qnn.QuantIdentity)
141             and hasattr(m, "fn")
142             and isinstance(m.fn, nn.Module)
143             and m.__class__.__name__.lower().startswith("residual")
144         ):
145             residuals.append(m)
146
147     def make_iq_hook(resid_mod):
148         buf = buffers[resid_mod]
149
150         def iq_hook(module, inputs, output):
151             in_qt = inputs[0] if (isinstance(inputs, tuple) and len(inputs) > 0) else None
152             s_in = _scale_of(in_qt)
153             s_q = _scale_of(output)
154             buf.append((s_in, s_q))
155         return iq_hook
156
157     def resid_hook(module, inputs, output):
158         # Expect two entries appended in order of calls:
159         #   first = fn(x) path, second = x path  (per your Residual.forward)
160         name = name_of.get(module, "<Residual>")
161         buf = buffers[module]
162
163         s_in_fn, s_q_fn = buf.popleft() if len(buf) else (None, None)
```

```
        s_in_x,  s_q_x  = buf.popleft() if len(buf) else (None, None)

        def code_shift(s_in, s_q):
            if s_in is None or s_q is None or s_in <= 0 or s_q <= 0:
                return 0
            # Signed CODE shift: +k => >>k,  -k => <<|k|
            return int(round(math.log2(s_q / s_in)))

        k_fn = code_shift(s_in_fn, s_q_fn)
        k_x  = code_shift(s_in_x,  s_q_x)

        writer.add_row(k_x, k_fn, comment=name)

    # Register hooks
    for resid in residuals:
        buffers[resid] = deque()
        hooks.append(resid.input_quant.register_forward_hook(make_iq_hook(resid)))
        hooks.append(resid.register_forward_hook(resid_hook))

    def cleanup():
        writer.close()
        for h in hooks:
            h.remove()

    return cleanup

def residual_scale_extractor(device, model_dir):
    config = load_yaml(os.path.join(model_dir, "config.yml"))
    model_pth = os.path.join(model_dir, "model.pth")
    model = load_model(config, model_pth, device)
    dummy = generate_input(config, device)

    # Warm-up pass to build quantized graph
    _ = model(dummy)

    # Collect residual shifts on another pass
    out_dir = os.path.join(model_dir, "weights")
    cleanup = attach_residual_hooks(model, out_dir)
    _ = model(dummy)
    cleanup()

    print(f"[OK] Wrote residual shifts to: {os.path.join(out_dir, 'scaleRES.h')}")
```

Listing C.5: Quantized Scale Export Code

C.6 Quantized Model Describing

```
from abc import ABC, abstractmethod
import argparse
import os

from brevitas.nn import QuantConv2d
import numpy as np
import torch
import yaml

from network import networks


def float_to_fixed_array(values, scale, dtype=np.int8):
    values = np.asarray(values, dtype=np.float32)
```

```
    scale = np.asarray(scale, dtype=np.float32)
    q = np.round(values / scale)
    return q.astype(dtype)


def bit_width_to_dtype(bit_width, signed=True):
    """
    Returns a NumPy integer dtype corresponding to the given bit width.
    Parameters:
        bit_width (int): Bit width of the desired integer type (8, 16, 32, 64).
        signed (bool): Whether to use signed integers. Default is True.
    Returns:
        np.dtype: Corresponding NumPy dtype (e.g., np.int8, np.uint16).
    Raises:
        ValueError: If the bit width is not supported.
    """
    dtype_map = {
        8:  (np.int8,  np.uint8),
        16: (np.int16, np.uint16),
        32: (np.int32, np.uint32),
        64: (np.int64, np.uint64),
    }

    if bit_width not in dtype_map:
        raise ValueError(f"Unsupported bit width: {bit_width}. Must be one of: {list(
            dtype_map.keys())}")
    return dtype_map[bit_width][0] if signed else dtype_map[bit_width][1]



def load_yaml(file_path) -> dict:
    with open(file_path) as file:
        return yaml.safe_load(file)


def setcache(m):
    m.cache_inference_quant_bias = True


def parse_args():
    parser = argparse.ArgumentParser()

    # base
    parser.add_argument(
        "--model_dir",
        type=str,
        help="Trained model directory (needs files: model.pth, config.yml)",
        required=True,
    )
    return vars(parser.parse_args())


def load_model(config, model_pth, device):
    """
    Load the trained model.
    """
    model = networks.get_model(model_name=config["model"], config=config).to(device)
    model.apply(setcache)
    state_dict = torch.load(model_pth, map_location=device, weights_only=True)
    model.load_state_dict(state_dict=state_dict, strict=False)
    return model.eval()


```

```python
def generate_input(config, device):
    C = config["input_channels"]
    H = config["input_h"]
    W = config["input_w"]
    return torch.zeros(1, C, H, W).to(device)


def array_2_string_cstyle(arr, idx, indent=4):
    """
    Convert a NumPy array to a nicely formatted C-style initializer (just the body, no declaration),
    with a C comment indicating the shape.
    """
    def format_array(a, level=1):
        spacing = ' ' * (indent * level)
        if a.ndim == 1:
            elements = ', '.join(f"{x:2}" for x in a)
            return spacing + "{ " + elements + "}"

        inner = ',\n'.join(format_array(sub, level+1) for sub in a)
        return spacing + "{\n" + inner + "\n" + spacing + "}"

    c_comment = f"/* idx: {idx}, shape: {arr.shape} */\n"
    return c_comment + format_array(arr, 0) + ","


def delete_if_found(file_path):
    if os.path.exists(file_path):
        os.remove(file_path)



class ParametersCppBase(ABC):
    def __init__(self, output_dir, weight_filename, bias_filename):
        self.weight_name = os.path.splitext(weight_filename)[0]
        self.bias_name = os.path.splitext(bias_filename)[0]
        os.makedirs(output_dir, exist_ok=True)

        self.weigths_file = os.path.join(output_dir, weight_filename)
        if os.path.exists(self.weigths_file):
            os.remove(self.weigths_file)

        self.bias_file = os.path.join(output_dir, bias_filename)
        if os.path.exists(self.bias_file):
            os.remove(self.bias_file)

        self.w_counter = 0
        self.b_counter = 0
        self.w_max_shape = None
        self.b_max_shape = None

    @abstractmethod
    def handle_weights(self, fixed_weights):
        """Transform the weights into the desired shape."""

    def _array_2_string_cstyle(self, arr, idx, indent=4):
        """
        Convert a NumPy array to a nicely formatted C-style initializer,
        with a C comment indicating index and shape.
        """
        def format_array(a, level=1):
            spacing = ' ' * (indent * level)
            if a.ndim == 1:
```

```
                elements = ', '.join(f"{x:2}" for x in a)
                return spacing + "{ " + elements + "}"

            inner = ',\n'.join(format_array(sub, level+1) for sub in a)
            return spacing + "{\n" + inner + "\n" + spacing + "}"

        c_comment = f"\n\n/* idx: {idx}, shape: {arr.shape} */\n"
        return c_comment + format_array(arr, 1) + ","

    def write_weights(self, fixed_weights):
        handled_weights = self.handle_weights(fixed_weights)
        c_string = self._array_2_string_cstyle(handled_weights, self.w_counter)
        with open(self.weigths_file, "a") as f:
            if self.w_counter == 0:
                self.w_max_shape = handled_weights.shape
                brackets = "[]" * handled_weights.ndim
                f.write("// Auto-generated from quantized weights model\n")
                f.write("// See end of map declaration to find max map size\n")
                f.write(f"const static int {self.weight_name}[]{brackets} = {{")
            else:
                self.w_max_shape = tuple(max(a, b) for a, b in zip(self.w_max_shape,
                    handled_weights.shape, strict=False))
            f.write(c_string)

        self.w_counter += 1

    def write_bias(self, fixed_bias):
        c_string = self._array_2_string_cstyle(fixed_bias, self.w_counter)
        with open(self.bias_file, "a") as f:
            if self.b_counter == 0:
                self.b_max_shape = fixed_bias.shape
                brackets = "[]" * fixed_bias.ndim
                f.write("// Auto-generated from quantized weights model\n")
                f.write("// See end of map declaration to find max map size\n")
                f.write(f"const static int {self.bias_name}[]{brackets} = {{")
            else:
                self.b_max_shape = tuple(max(a, b) for a, b in zip(self.b_max_shape,
                    fixed_bias.shape, strict=False))
            f.write(c_string)

        self.b_counter += 1

    def cleanup(self):
        if self.w_counter > 0:
            with open(self.weigths_file, "a") as f:
                f.write("\n};\n")
                dims_brackets = ''.join(f'[{d}]' for d in self.w_max_shape)
                f.write(f"// [{self.w_counter}]{dims_brackets} \n")

        if self.b_counter > 0:
            with open(self.bias_file, "a") as f:
                f.write("\n};\n")
                dims_brackets = ''.join(f'[{d}]' for d in self.b_max_shape)
                f.write(f"// [{self.w_counter}]{dims_brackets} \n")

class ParametersPWCpp(ParametersCppBase):
    def __init__(self, output_dir):
        super().__init__(output_dir, "weightsPW.h", "biasPW.h")

    def handle_weights(self, fixed_weights):
        # (outC, inC, 1, 1) -> (outC, inC)
        return np.squeeze(fixed_weights, axis=(2, 3))

```

```
200
201 class ParametersDWCpp(ParametersCppBase):
202     def __init__(self, output_dir):
203         super().__init__(output_dir, "weightsDW.h", "biasDW.h")
204
205     def handle_weights(self, fixed_weights):
206         # (outC, 1, kH, kW) -> (kH, kW, outC)
207         return np.squeeze(fixed_weights, axis=1).transpose(1, 2, 0)
208
209 class Parameters3DCpp(ParametersCppBase):
210     def __init__(self, output_dir):
211         super().__init__(output_dir, "weights3D.h", "bias3D.h")
212
213     def handle_weights(self, fixed_weights):
214         # (outC, inC, kH, kW) -> (kH, kW, outC, inC)
215         return fixed_weights.transpose(2, 3, 0, 1)
216
217 # class ParametersG2Cpp(ParametersCppBase):
218 #     def __init__(self, output_dir):
219 #         super().__init__(output_dir, "weightsG2.h", "biasG2.h")
220
221 #     def handle_weights(self, fixed_weights):
222 #         # (outC, inC, kH, kW) -> (kH, kW, outC, inC)
223 #         return  fixed_weights.transpose(2, 3, 0, 1)
224
225
226
227 def extract_weights_and_bias(model, output_dir, device):
228     os.makedirs(output_dir, exist_ok=True)
229     id = 0
230
231     weightsDWCpp = ParametersDWCpp(output_dir)
232     weightsPWCpp = ParametersPWCpp(output_dir)
233     # weightsG2Cpp = ParametersG2Cpp(output_dir)
234     weights3DCpp = Parameters3DCpp(output_dir)
235     weights_cpp_class : ParametersCppBase = None
236
237     for name, module in model.named_modules():
238         base_name = f"{id}_{name}_{module.__class__.__name__}"
239         description_file_name = f"{base_name}.txt"
240         debug_file_name =  f"{base_name}.values"
241         bin_file_name = f"{base_name}.bin"
242         description_file = os.path.join(output_dir, description_file_name)
243         bin_file = os.path.join(output_dir, bin_file_name)
244         debug_file = os.path.join(output_dir, debug_file_name)
245
246         delete_if_found(description_file)
247         delete_if_found(bin_file)
248         delete_if_found(debug_file)
249
250         if isinstance(module, QuantConv2d):
251
252             if module.groups == module.in_channels and module.in_channels == module.
                out_channels:
253                 weights_cpp_class = weightsDWCpp
254             elif module.kernel_size == (1, 1) and module.groups == 1:
255                 weights_cpp_class = weightsPWCpp
256             # elif module.groups == 2:
257                 # weights_cpp_class = weightsG2Cpp
258             else:
259                 weights_cpp_class = weights3DCpp
260
261
```

```
            # === Weights ===
            if hasattr(module, "weight") and hasattr(module, "weight_quant"):
                id+=1
                quant_tensor = module.quant_weight()
                scale = getattr(quant_tensor, "scale", None)
                scale = scale.item() if hasattr(scale, 'item') else scale
                signed = getattr(quant_tensor, "signed_t", None)
                weights = getattr(quant_tensor, "value", None)
                bit_width = getattr(quant_tensor, "bit_width", None)
                dtype = bit_width_to_dtype(int(bit_width), signed)

                weights_np = weights.detach().to(device).numpy()
                fixed_values = float_to_fixed_array(weights_np, scale, dtype)

                with open(description_file, "a") as f:
                    f.write("#" * 80 + "\n")
                    f.write(f"[Weight] Layer: {name} ({module.__class__.__name__})\n")
                    f.write(f"  Shape: {tuple(quant_tensor.shape)}\n")
                    f.write(f"  Bit Width: {bit_width.item()}\n")
                    f.write(f"  Signed: {signed}\n")
                    f.write(f"  Scale: {scale}\n")
                    f.write(
                        f"  Sample Weights: {np.array2string(weights_np[:10], precision=5)
                            }...\n"
                    )
                    f.write("-" * 80 + "\n")

                with open(debug_file, "a") as f:
                    f.write(np.array2string(weights_np, precision=5))
                    f.write("\nFixed Point: \n" + np.array2string(fixed_values))

                with open(bin_file, "ab") as f:
                    f.write(fixed_values.tobytes())

                # ----------------------------------C FILE STORE
                    ------------------------------------
                weights_cpp_class.write_weights(fixed_values)

            # === Bias ===
            if hasattr(module, "bias_quant"):
                quant_tensor = module.quant_bias()
                scale = getattr(quant_tensor, "scale", None)
                scale = scale.item() if hasattr(scale, 'item') else scale
                signed = getattr(quant_tensor, "signed_t", None)
                bias = getattr(quant_tensor, "value", None)
                bit_width = getattr(quant_tensor, "bit_width", None)

                bias_np = bias.detach().to(device).numpy()
                array_flat = bias_np.flatten(order="C")

                with open(description_file, "a") as f:
                    f.write(f"[Bias] Layer: {name} ({module.__class__.__name__})\n")
                    f.write(f"  Shape: {tuple(quant_tensor.shape)}\n")
                    f.write(f"  Bit Width: {bit_width.item()}\n")
                    f.write(f"  Signed: {signed}\n")
                    f.write(f"  Scale: {scale}\n")
                    f.write(
                        f"  Sample Bias: {np.array2string(array_flat[:10], precision=5)}...\n
                            "
                    )
                    f.write("#" * 80 + "\n\n")

                dtype = bit_width_to_dtype(int(bit_width), signed)
```

```
                fixed_values = float_to_fixed_array(array_flat, scale, dtype)

                with open(debug_file, "a") as f:
                    f.write("\n" + "-" * 80 + "\n")
                    f.write(np.array2string(array_flat, precision=5))
                    f.write("\nFixed Point: \n" + np.array2string(fixed_values))

                with open(bin_file, "ab") as f:
                    f.write(fixed_values.tobytes())

                # ----------------------------------C FILE STORE
                    ------------------------------------
                weights_cpp_class.write_bias(fixed_values)


    # DONT FORGET
    weightsDWCpp.cleanup()
    weightsPWCpp.cleanup()
    weights3DCpp.cleanup()
    # weightsG2Cpp.cleanup()


def weight_extractor(device, model_dir):
    config = load_yaml(os.path.join(model_dir, "config.yml"))
    model_pth = os.path.join(model_dir, "model.pth")
    model = load_model(config, model_pth, device)

    output_dir = os.path.join(model_dir, "weights")

    input_ = generate_input(config, device)
    model(input_)  # Run a dummy inference first
    extract_weights_and_bias(model, output_dir, device)
```

Listing C.6: Quantized Weights Extraction Code

D High Level Synthesis (HLS) Code

D.1 Pointwise Core HLS Code

```
// layerPW.cpp
#include "layerPW.h"

void HW_pointWise(
    hls::stream<bus64_t>& strm_in,
    hls::stream<ap_uint<64>>& fifo_outA0,
    hls::stream<ap_uint<64>>& fifo_outA1,
    hls::stream<ap_uint<64>>& fifo_outB0,
    hls::stream<ap_uint<64>>& fifo_outB1,
    ap_uint<32> config
) {
#pragma HLS interface axis port=strm_in

    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    unsigned int layer_ID = config.range(13,9).to_uint();       // layer identifier
    unsigned int channel = config.range(17,14).to_uint();      // in channels
    ap_uint<6> filters = config.range(23,18);      // out channels
    ap_uint<1> lastLayer = config.range(24,24);    // last layer flag

    const int scale = scalePW[layer_ID].to_int();

    acc_t acc1[8], acc2[8];
    #pragma HLS ARRAY_PARTITION variable=acc1 complete
    #pragma HLS ARRAY_PARTITION variable=acc2 complete

    ap_uint<64> line_buffer[2][256*4];

    bool fifoSel = false;

    // Process two lines at a time
    process_pairs: for (ap_uint<9> i = 0; i < map_size; i += 2) {
        #pragma HLS LOOP_TRIPCOUNT max=128

        for (int l = 0; l < 2; l++) {
            #pragma HLS UNROLL
            for (int j = 0; j < map_size * channel; j++) {
                #pragma HLS LOOP_TRIPCOUNT max=(256*4) min=1
                #pragma HLS PIPELINE II=1
                bus64_t value = strm_in.read();
                line_buffer[l][j].range(63, 0) = value.data.range(63, 0);
            }
        }

        // Process the two loaded lines
        map_col_opt: for (int j = 0; j < map_size; j++) {
            #pragma HLS LOOP_TRIPCOUNT max=256
```

```
47
48              ft_opt: for (int fx = 0; fx < filters; fx++) {
49                  #pragma HLS LOOP_TRIPCOUNT max=8
50
51                  bias_init: for (int f = 0; f < 8; f++) {
52                      #pragma HLS UNROLL
53                      mult_t bias = biasPW[layer_ID][fx*8+f];
54                      acc1[f] = bias;
55                      acc2[f] = bias;
56                  }
57
58                  // Process input channels
59                  ch_opt: for (int ch_idx = 0; ch_idx < channel; ch_idx++) {
60                      #pragma HLS LOOP_TRIPCOUNT max=8
61                      #pragma HLS PIPELINE
62
63                      int map_idx = (j * channel) + ch_idx;
64                      ap_uint<64> data1 = line_buffer[0][map_idx];
65                      ap_uint<64> data2 = line_buffer[1][map_idx];
66
67                      // Unroll both data processing and MAC completely
68                      data_mac: for (int c = 0; c < 8; c++) {
69                          #pragma HLS UNROLL
70                          data_t d1 = data1.range(c*8+7, c*8);
71                          data_t d2 = data2.range(c*8+7, c*8);
72
73                          filters_mac: for (int f = 0; f < 8; f++) {
74                              #pragma HLS UNROLL
75                              data_t weight = weightsPW[layer_ID][fx*8+f][ch_idx*8+c];
76                              acc1[f] += d1 * weight;
77                              acc2[f] += d2 * weight;
78                          }
79                      }
80                  }
81
82                  // Scale, pack and output in one step
83                  ap_uint<64> out1 = 0, out2 = 0;
84                  output_pack: for (int p = 0; p < 8; p++) {
85                      #pragma HLS UNROLL
86                      if (lastLayer) {
87                          // sigmoid+0.5 threshold == sign test on logits
88                          data_t b1 = (acc1[p] >= 0);
89                          data_t b2 = (acc2[p] >= 0);
90
91                          out1.range(p*8+7, p*8) = b1.range(7, 0);
92                          out2.range(p*8+7, p*8) = b2.range(7, 0);
93                      } else {
94                          // normal hidden layers: ReLU + clamp to int8
95                          data_t v1 = (data_t)CLAMP8_RELU_AP(acc1[p] >> scale);
96                          data_t v2 = (data_t)CLAMP8_RELU_AP(acc2[p] >> scale);
97                          out1.range(p*8+7, p*8) = v1.range(7, 0);
98                          out2.range(p*8+7, p*8) = v2.range(7, 0);
99                      }
100                 }
101
102                 // Alternate FIFO writes
103                 if (fifoSel) {
104                     fifo_outA1.write(out1);
105                     fifo_outB1.write(out2);
106                 } else {
107                     fifo_outA0.write(out1);
108                     fifo_outB0.write(out2);
109                 }
```

```
                fifoSel = !fifoSel;
            }
        }
    }
}

// CORRECT
void HW_writeData(
    hls::stream<bus64_t>& strm_out,
    hls::stream<ap_uint<64>>& fifo_outA0,
    hls::stream<ap_uint<64>>& fifo_outA1,
    hls::stream<ap_uint<64>>& fifo_outB0,
    hls::stream<ap_uint<64>>& fifo_outB1,
    config_t config
){
#pragma HLS interface axis port=strm_out

    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    ap_uint<6> filters = config.range(23,18);      // out channels
    ap_uint<16> total_iterations = (map_size * filters) >> 1; // divide by 2
    ap_uint<16> last_iteration = total_iterations - 1;
    ap_uint<1> lastLayer = config.range(24,24);    // last layer flag

    bus64_t bus_template;
    bus_template.keep = 0xFF;
    bus_template.strb = 0xFF;
    bus_template.last = 0;

    if (lastLayer) {
        // Emit HxWx1 in HWC order: 8 consecutive pixels (same row) per 64-bit word.
        const ap_uint<16> words_per_row = map_size >> 3; // 8 pixels per word
        const ap_uint<16> total_output_words = (map_size * map_size) >> 3;

        ap_uint<16> word_count = 0;

        // process rows in pairs (A FIFOs = row y, B FIFOs = row y+1)
        for (ap_uint<9> y = 0; y < map_size; y += 2) {
        #pragma HLS LOOP_TRIPCOUNT max=256
            // --- Row y (use A0/A1 only) ---
            for (ap_uint<16> x = 0; x < words_per_row; ++x) {
            #pragma HLS LOOP_TRIPCOUNT max=32
            #pragma HLS PIPELINE II=4
                ap_uint<64> w1 = fifo_outA0.read();
                ap_uint<64> w2 = fifo_outA1.read();
                ap_uint<64> w3 = fifo_outA0.read();
                ap_uint<64> w4 = fifo_outA1.read();
                ap_uint<64> w5 = fifo_outA0.read();
                ap_uint<64> w6 = fifo_outA1.read();
                ap_uint<64> w7 = fifo_outA0.read();
                ap_uint<64> w8 = fifo_outA1.read();

                ap_uint<64> out_word = 0;
                out_word.range( 7, 0) = w1.range(7, 0);
                out_word.range(15, 8) = w2.range(7, 0);
                out_word.range(23,16) = w3.range(7, 0);
                out_word.range(31,24) = w4.range(7, 0);
                out_word.range(39,32) = w5.range(7, 0);
                out_word.range(47,40) = w6.range(7, 0);
                out_word.range(55,48) = w7.range(7, 0);
                out_word.range(63,56) = w8.range(7, 0);

                bus64_t out = bus_template;
                out.data = out_word;
```

```
                out.last = (word_count == total_output_words - 1) ? 1 : 0;
                strm_out.write(out);
                ++word_count;
            }

            // --- Row y+1 (use B0/B1 only) ---
            if (y + 1 < map_size) {
                for (ap_uint<16> x = 0; x < words_per_row; ++x) {
                #pragma HLS LOOP_TRIPCOUNT max=32
                #pragma HLS PIPELINE II=4
                    ap_uint<64> w1 = fifo_outB0.read();
                    ap_uint<64> w2 = fifo_outB1.read();
                    ap_uint<64> w3 = fifo_outB0.read();
                    ap_uint<64> w4 = fifo_outB1.read();
                    ap_uint<64> w5 = fifo_outB0.read();
                    ap_uint<64> w6 = fifo_outB1.read();
                    ap_uint<64> w7 = fifo_outB0.read();
                    ap_uint<64> w8 = fifo_outB1.read();

                    ap_uint<64> out_word = 0;
                    out_word.range( 7, 0) = w1.range(7, 0);
                    out_word.range(15, 8) = w2.range(7, 0);
                    out_word.range(23,16) = w3.range(7, 0);
                    out_word.range(31,24) = w4.range(7, 0);
                    out_word.range(39,32) = w5.range(7, 0);
                    out_word.range(47,40) = w6.range(7, 0);
                    out_word.range(55,48) = w7.range(7, 0);
                    out_word.range(63,56) = w8.range(7, 0);

                    bus64_t out = bus_template;
                    out.data = out_word;
                    out.last = (word_count == total_output_words - 1) ? 1 : 0;
                    strm_out.write(out);
                    ++word_count;
                }
            }
        }
    } else {
        LOOP_Y: for (ap_uint<9> y = 0; y < map_size; y++) {
            #pragma HLS LOOP_TRIPCOUNT max=256

            const bool use_A_fifos = (y & 1) == 0;
            const bool is_last_row = (y == map_size - 1);

            LOOP_X: for (ap_uint<16> x = 0; x < total_iterations; x++) {
                #pragma HLS LOOP_TRIPCOUNT max=512 //(256*4)/2
                #pragma HLS PIPELINE II=2

                // Use local variables to help with timing
                bus64_t tmp1 = bus_template;
                bus64_t tmp2 = bus_template;

                // Conditional read based on pre-calculated flag
                if (use_A_fifos) {
                    tmp1.data = fifo_outA0.read();
                    tmp2.data = fifo_outA1.read();
                } else {
                    tmp1.data = fifo_outB0.read();
                    tmp2.data = fifo_outB1.read();
                }

                const bool is_last_element = (is_last_row && (x == last_iteration));
                tmp2.last = is_last_element ? 1 : 0;
```

```
                strm_out.write(tmp1);
                strm_out.write(tmp2);
            }
        }
    }
}



void HW_layerPW(
    hls::stream<bus64_t> &strm_in,
    hls::stream<bus64_t> &strm_out,
    config_t config
) {
    #pragma HLS interface axis port=strm_in
    #pragma HLS interface axis port=strm_out
    #pragma HLS INTERFACE s_axilite port=return bundle=AXILite
    #pragma HLS INTERFACE s_axilite port=config bundle=AXILite
    #pragma HLS DATAFLOW

    static hls::stream<ap_uint<64>> fifoA0("fifoA0");
    static hls::stream<ap_uint<64>> fifoA1("fifoA1");
    static hls::stream<ap_uint<64>> fifoB0("fifoB0");
    static hls::stream<ap_uint<64>> fifoB1("fifoB1");

#pragma HLS STREAM variable=fifoA0 depth=1024
#pragma HLS STREAM variable=fifoA1 depth=1024
#pragma HLS STREAM variable=fifoB0 depth=1024
#pragma HLS STREAM variable=fifoB1 depth=1024


    HW_pointWise(strm_in, fifoA0, fifoA1, fifoB0, fifoB1, config);
    HW_writeData(strm_out, fifoA0, fifoA1, fifoB0, fifoB1, config);
}
```

Listing D.1: Pointwise Core HLS Code

D.2 Depthwise Core HLS Code

```
// layerDW.cpp
// #include "layerDW.h"

#define MAX(a, b) (((data_t)(a) > (data_t)(b)) ? (a) : (b))

void HW_readData(
    hls::stream<bus64_t> &strm_in,
    hls::stream<ap_uint<64>> &fifo_out,
    hls::stream<ap_uint<64>> &res_con,
    ap_uint<32> config
) {
#pragma HLS interface axis port=strm_in

    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    ap_uint<2> channel = config.range(15,14);
    ap_uint<3> pad = config.range(22,20);
    ap_uint<1> maxpool = config.range(23,23);


    ap_uint<9> map_size_maxpool = map_size >> maxpool;
```

```
21
22
23      ap_uint<64> buffer0_ch0[128], buffer0_ch1[128];
24      ap_uint<64> buffer1_ch0[128], buffer1_ch1[128];
25
26
27      ap_uint<64> zeros = 0;
28      bus64_t tmp1, tmp2;
29      ap_uint<64> buf[4];
30  #pragma HLS ARRAY_PARTITION variable=buf type=complete dim=1
31
32      // Write top padding
33      paddTop: for (ap_uint<9> i = 0; i < (map_size_maxpool + pad*2) * channel * pad; i++) {
34          #pragma HLS LOOP_TRIPCOUNT max=(256+2)
35          fifo_out.write(zeros);
36      }
37
38      // Process data rows
39      rd_line: for (ap_uint<9> i = 0; i < map_size; i++) {
40          #pragma HLS LOOP_TRIPCOUNT max=256
41
42          bool write_row = (maxpool == 0) || (i % 2 == 0);
43          bool process_maxpool = (maxpool == 1) && (i % 2 == 1);
44
45          // Left padding
46          if (pad > 0 && write_row) {
47              paddLeft: for (ap_uint<4> j = 0; j < pad * channel; j++) {
48                  #pragma HLS LOOP_TRIPCOUNT max=8
49                  fifo_out.write(zeros);
50              }
51          }
52
53          // Process map data
54          if (channel == 1) {
55              // Single channel processing (unchanged)
56              rd_map: for (int j = 0; j < map_size; j += 2) {
57                  #pragma HLS LOOP_TRIPCOUNT max=128
58                  #pragma HLS PIPELINE II=2
59
60                  strm_in.read(tmp1);
61                  strm_in.read(tmp2);
62
63                  if (maxpool == 0) {
64                      fifo_out.write(tmp1.data);
65                      fifo_out.write(tmp2.data);
66                      res_con.write(tmp1.data);
67                      res_con.write(tmp2.data);
68                  } else {
69                      int idx = j / 2;
70
71                      if ((i & 1) == 0) {
72                          buffer0_ch0[idx] = tmp1.data.range(63, 0);
73                          buffer1_ch0[idx] = tmp2.data.range(63, 0);
74                      } else {
75                          for (ap_uint<4> k = 0; k < 8; k++) {
76                              #pragma HLS UNROLL
77                              data_t curr1 = tmp1.data.range(k*8+7, k*8);
78                              data_t curr2 = tmp2.data.range(k*8+7, k*8);
79                              data_t prev1 = buffer0_ch0[idx].range(k*8+7, k*8);
80                              data_t prev2 = buffer1_ch0[idx].range(k*8+7, k*8);
81
82                              data_t max_val = MAX(MAX(prev1, prev2), MAX(curr1, curr2));
83                              buffer0_ch0[idx].range(k*8+7, k*8) = max_val;
```

```
                    }

                    fifo_out.write(buffer0_ch0[idx]);
                    res_con.write(buffer0_ch0[idx]);
                }
            }
        }
    } else {
        // Multi-channel processing with separate channel buffers
        rd_map2: for (int j = 0; j < map_size; j += 2) {
            #pragma HLS LOOP_TRIPCOUNT max=16
            #pragma HLS PIPELINE II=4

            // Read 4 data elements for 2 channels
            buf[0] = strm_in.read().data;
            buf[1] = strm_in.read().data;
            buf[2] = strm_in.read().data;
            buf[3] = strm_in.read().data;

            if (maxpool == 0) {
                fifo_out.write(buf[0]);
                fifo_out.write(buf[1]);
                fifo_out.write(buf[2]);
                fifo_out.write(buf[3]);

                res_con.write(buf[0]);
                res_con.write(buf[1]);
                res_con.write(buf[2]);
                res_con.write(buf[3]);
            } else {
                int idx = j / 2;

                if (i % 2 == 0) {
                    // Store first row data in separate channel buffers
                    buffer0_ch0[idx] = buf[0];
                    buffer0_ch1[idx] = buf[1];
                    buffer1_ch0[idx] = buf[2];
                    buffer1_ch1[idx] = buf[3];
                } else {
                    // Process max pooling with separate buffers (no conflicts)
                    ap_uint<64> result0 = 0, result1 = 0;

                    maxpool_ch0: for (ap_uint<4> k = 0; k < 8; k++) {
                        #pragma HLS UNROLL
                        data_t max0 = MAX(buffer0_ch0[idx].range(k*8+7, k*8), buf[0].
                            range(k*8+7, k*8));
                        data_t max1 = MAX(buffer1_ch0[idx].range(k*8+7, k*8), buf[2].
                            range(k*8+7, k*8));
                        result0.range(k*8+7, k*8) = MAX(max0, max1);
                    }

                    maxpool_ch1: for (ap_uint<4> k = 0; k < 8; k++) {
                        #pragma HLS UNROLL
                        data_t max0 = MAX(buffer0_ch1[idx].range(k*8+7, k*8), buf[1].
                            range(k*8+7, k*8));
                        data_t max1 = MAX(buffer1_ch1[idx].range(k*8+7, k*8), buf[3].
                            range(k*8+7, k*8));
                        result1.range(k*8+7, k*8) = MAX(max0, max1);
                    }

                    if (process_maxpool) {
                        fifo_out.write(result0);
                        fifo_out.write(result1);
```

```
                            res_con.write(result0);
                            res_con.write(result1);
                        }
                    }
                }
            }
        }

        // Right padding
        if (pad > 0 && ((maxpool == 0) || (i % 2 == 1))) {
            paddRight: for (ap_uint<4> j = 0; j < pad * channel; j++) {
                #pragma HLS LOOP_TRIPCOUNT max=8
                fifo_out.write(zeros);
            }
        }
    }

    // Write bottom padding
    paddBottom: for (ap_uint<9> i = 0; i < (map_size_maxpool + pad*2) * channel * pad; i++) {
        #pragma HLS LOOP_TRIPCOUNT max=(32+2)*2
        fifo_out.write(zeros);
    }
}




/*
VITIS HLS CONV2D DEPTHWISE IMPLEMENTATION
OUTPUT DIM SHOULD BE SAME AS INPUT
*/
#define MAX_LINES 10
void HW_depthWise(
        hls::stream<ap_uint<64>> &fifo_in,
        hls::stream<ap_uint<256>> &fifo_out1, // convolution output
        hls::stream<ap_uint<64>> &res_con, // residual connection
        config_t config
    )
{
    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    unsigned int layer_ID = config.range(13,9).to_uint();       // layer identifier
    unsigned int channel = config.range(15,14).to_uint();
    ap_uint<4> kernel_size = config.range(19,16);
    ap_uint<3> pad = config.range(22,20);
    ap_uint<1> maxpool = config.range(23,23);


    const int scale = scaleDW[layer_ID].to_int();


    ap_uint<64> mapa_inDW[MAX_LINES][258];
#pragma HLS BIND_STORAGE variable=mapa_inDW type=ram_s2p impl=lutram // important, use LUTs
    instead of BRAMs

    ap_uint<64> tmp1, tmp2;

    ap_uint<9> map_size_maxpool = map_size >> maxpool;

    ap_uint<17> maxReads = ((map_size_maxpool+2*pad) * (map_size_maxpool+2*pad)) * channel;
        // correct
    ap_uint<17> readCounter = 0;

```

```
204     ap_uint<5> initialLines;
205     if (kernel_size == 3) initialLines = 4;
206     if (kernel_size == 7) initialLines = 8;
207     if (kernel_size == 9) initialLines = 10;
208 
209     int cl = (channel == 1) ? 4 : 2;
210     int ck = (channel == 1) ? 0 : 8;
211 
212     // Doing 4 accumulations per channel at a time
213     acc_t acc1[8], acc2[8], acc3[8], acc4[8];
214     ap_uint<256> buffer;
215 
216     rd_k_lines: for (int i = 0; i < initialLines; i++){
217         #pragma HLS LOOP_TRIPCOUNT max=10
218         for (int j  = 0; j < (map_size_maxpool+2*pad)*channel; j++){
219             #pragma HLS LOOP_TRIPCOUNT max=1024
220             tmp1 = fifo_in.read();
221             readCounter++;
222             mapa_inDW[i][j].range(63,0) = tmp1.range(63,0);
223         }
224     }
225 
226     map_line: for (int i = 0; i < map_size_maxpool; i++) {
227     #pragma HLS LOOP_TRIPCOUNT max=256
228 
229         map_col: for (int j = 0; j < map_size_maxpool; j+=cl) { // columns (step by cl = 4 or
                 2)
230             #pragma HLS LOOP_TRIPCOUNT max=64
231 
232             bias_init: for(int c = 0; c < 8; c++) {
233                 #pragma HLS UNROLL
234                 mult_t b1 = biasDW[layer_ID][c];
235                 mult_t b2 = biasDW[layer_ID][c+ck];
236                 acc1[c] = b1;
237                 acc2[c] = b2;
238                 acc3[c] = b1;
239                 acc4[c] = b2;
240             }
241 
242             k1a: for (int ki = 0; ki < kernel_size; ki++) { // k-height
243                 #pragma HLS LOOP_TRIPCOUNT max=9
244                 int  ii = i + ki; // correct
245 
246                 k2a: for (int  kj = 0; kj < kernel_size; kj++) { // k-width
247                     #pragma HLS LOOP_TRIPCOUNT max=9
248                     #pragma HLS PIPELINE
249 
250                     int  jj = (j + kj) * channel;
251                     ch1: for (int c = 0; c < 8; c++){ // process 8 channels in parallel and 4
                             accums at a time. that means that 64 * 4 == 256
252                         data_t v1 = mapa_inDW[(ii%MAX_LINES)][(jj+0)].range(c*8+7, c*8);
253                         data_t v2 = mapa_inDW[(ii%MAX_LINES)][(jj+1)].range(c*8+7, c*8);
254                         data_t v3 = mapa_inDW[(ii%MAX_LINES)][(jj+2)].range(c*8+7, c*8);
255                         data_t v4 = mapa_inDW[(ii%MAX_LINES)][(jj+3)].range(c*8+7, c*8);
256 
257                         data_t w1 = weightsDW[layer_ID][ki][kj][c];
258                         data_t w2 = weightsDW[layer_ID][ki][kj][c+ck];
259 
260                         acc1[c] += v1*w1;
261                         acc2[c] += v2*w2;
262                         acc3[c] += v3*w1;
263                         acc4[c] += v4*w2;
264                     }
```

```
                }
            }

            relu: for (int c = 0; c < 8; c++){
                #pragma HLS UNROLL
                data_t v1 = (data_t)CLAMP8_RELU_AP(acc1[c] >> scale);
                data_t v2 = (data_t)CLAMP8_RELU_AP(acc2[c] >> scale);
                data_t v3 = (data_t)CLAMP8_RELU_AP(acc3[c] >> scale);
                data_t v4 = (data_t)CLAMP8_RELU_AP(acc4[c] >> scale);
                buffer.range(c*8+7,c*8) = v1.range(7,0);
                buffer.range(c*8+7+64,c*8+64) = v2.range(7,0);
                buffer.range(c*8+7+128,c*8+128) = v3.range(7,0);
                buffer.range(c*8+7+192,c*8+192) = v4.range(7,0);
            }

            fifo_out1.write(buffer);
        }
        // read 1 line
        if(readCounter < maxReads) {
            int line = (i + initialLines);
            for (int x = 0; x < (map_size_maxpool+2*pad)*channel; x++){
                #pragma HLS LOOP_TRIPCOUNT max=1024
                tmp1 = fifo_in.read();
                readCounter++;
                mapa_inDW[(line%MAX_LINES)][x].range(63,0) = tmp1.range(63,0);
            }
        }
    }
}



void HW_writeData(
      hls::stream<bus64_t> &strm_out,
      hls::stream<ap_uint<256>> &fifo_in1,
      hls::stream<ap_uint<64>> &res_con,
      ap_uint<32> config
) {
#pragma HLS interface axis port=strm_out

    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    unsigned int layer_ID = config.range(13,9).to_uint();       // layer identifier
    ap_uint<2> channel = config.range(15,14);
    ap_uint<1> maxpool = config.range(23,23);

    ap_uint<9> map_size_maxpool = map_size >> maxpool;

    ap_uint<128> buffer[128];
    ap_uint<64> buf, tmpB0, tmpB1;
    ap_uint<256> tmp1, tmp2;
    bus64_t tmpo;


    if (channel == 1){
        wr_line: for (int i = 0; i < map_size_maxpool*map_size_maxpool/4; i++){
            #pragma HLS LOOP_TRIPCOUNT max=16384
            #pragma HLS PIPELINE II=4

            tmp1 = fifo_in1.read();

            tmpo.keep = 0xFF;
```

```
            tmpo.strb = 0xFF;

            for (ap_uint<3> j = 0; j < 4; j++){ // 256/8 == 4
                tmpB0 = res_con.read(); // read 1 more

                for (ap_uint<4> c = 0; c < 8; c++) {
                    data_t v1 = tmp1.range(c*8+7 + j*64, c*8 + j*64);
                    data_t v2 = tmpB0.range(c*8+7, c*8);

                    v1 = safe_shift_ap(v1, scaleRES[layer_ID][1]);
                    v2 = safe_shift_ap(v2, scaleRES[layer_ID][0]);

                    ap_int<9> sum = v1 + v2;
                    if (sum > 127) sum = 127;
                    if (sum < -128) sum = -128;

                    buf.range(8*c+7, 8*c) = sum.range(7,0);
                }
                if (i == map_size_maxpool*map_size_maxpool/4-1 && j == 4-1)
                    tmpo.last = 1;
                else
                    tmpo.last = 0;

                tmpo.data.range(63,0) = buf.range(63,0);
                strm_out.write(tmpo);
            }
        }
    }
    else{
        wr_line2: for (ap_uint<11> i = 0; i < map_size_maxpool*map_size_maxpool/4; i++) {
            #pragma HLS LOOP_TRIPCOUNT max=1024
            #pragma HLS PIPELINE II=8

            // --- Drain residuals EARLY ---
            ap_uint<64> r0[4], r1[4];
            #pragma HLS ARRAY_PARTITION variable=r0 complete
            #pragma HLS ARRAY_PARTITION variable=r1 complete
            for (int j=0;j<4;j++) { r0[j] = res_con.read(); }  // for first 256b
            for (int j=0;j<4;j++) { r1[j] = res_con.read(); }  // for second 256b

            // Now fetch conv results
            ap_uint<256> t0 = fifo_in1.read();
            ap_uint<256> t1 = fifo_in1.read();

            bus64_t o = {0}; o.keep = 0xFF; o.strb = 0xFF;

            // first 256b -> 4x64b using r0[]
            for (int j=0;j<4;j++) {
                ap_uint<64> buf = 0;
                for (int c=0;c<8;c++) {
                    data_t v1 = t0.range(c*8+7 + j*64, c*8 + j*64);
                    data_t v2 = r0[j].range(c*8+7, c*8);

                    v1 = safe_shift_ap(v1, scaleRES[layer_ID][1]);
                    v2 = safe_shift_ap(v2, scaleRES[layer_ID][0]);

                    ap_int<9> s = v1 + v2;

                    if (s > 127) s = 127; else if (s < -128) s = -128;
                    buf.range(8*c+7, 8*c) = s.range(7,0);
                }
                o.data = buf;
                o.last = 0;
```

```
                strm_out.write(o);
            }

            // second 256b -> 4x64b using r1[]
            for (int j=0;j<4;j++) {
                ap_uint<64> buf = 0;
                for (int c=0;c<8;c++) {
                    data_t v1 = t1.range(c*8+7 + j*64, c*8 + j*64);
                    data_t v2 = r1[j].range(c*8+7, c*8);

                    v1 = safe_shift_ap(v1, scaleRES[layer_ID][1]);
                    v2 = safe_shift_ap(v2, scaleRES[layer_ID][0]);

                    ap_int<9> s = v1 + v2;

                    if (s > 127) s = 127; else if (s < -128) s = -128;
                    buf.range(8*c+7, 8*c) = s.range(7,0);
                }
                o.data = buf;
                o.last = (i == map_size_maxpool*map_size_maxpool/4-1 && j == 3) ? 1 : 0;
                strm_out.write(o);
            }
        }
    }
}




void HW_layerDW(
    hls::stream<bus64_t> &strm_in,
    hls::stream<bus64_t> &strm_out,
    config_t config
) {
#pragma HLS interface axis port=strm_in
#pragma HLS interface axis port=strm_out
#pragma HLS INTERFACE s_axilite port=return bundle=AXILite
#pragma HLS INTERFACE s_axilite port=config bundle=AXILite
#pragma HLS DATAFLOW

static hls::stream<ap_uint<64>> f1("f1");
static hls::stream<ap_uint<256>> f2("f2");
static hls::stream<ap_uint<64>> r_fifo("r_con");

#pragma HLS STREAM variable=f1 depth=2048       // Guaranteed startup buffering
#pragma HLS STREAM variable=f2 depth=2048        // Ample rate mismatch buffer
#pragma HLS STREAM variable=r_fifo depth=4096   // Safe residual buffering

    HW_readData(strm_in, f1, r_fifo, config);
    HW_depthWise(f1, f2, r_fifo, config);
    HW_writeData(strm_out, f2, r_fifo, config);
}
```

Listing D.2: Depthwise Core HLS Code

D.3 Convolution 3D Core HLS Code

```
#include "layerC3D.h"

void HW_upsample(
    hls::stream<bus64_t> &strm_in,
```

```
    hls::stream<bus64_t> &skip_con_in,
    hls::stream<ap_uint<64>> &fifo_out,
    config_t config
) {
#pragma HLS interface axis port=strm_in
#pragma HLS interface axis port=skip_con_in
#pragma HLS INLINE off

    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    unsigned int layer_ID = config.range(13,9).to_uint();       // layer identifier
    unsigned int channel = config.range(15,14).to_uint();       // channel
    ap_uint<1> upsample = config.range(18,18);
    ap_uint<1> firstLayer = config.range(19,19);
    ap_uint<1> skipCon = config.range(20,20);



    // 4-line rolling buffer; second dim sized to your worst-case (map_size*channel <= 128)
    ap_uint<64> mapa_inC2D[4][128];
#pragma HLS ARRAY_PARTITION variable=mapa_inC2D complete dim=1
#pragma HLS bind_storage  variable=mapa_inC2D type=RAM_2P impl=bram

    int iy0, ix0;
    int iy1, ix1;
    ap_int<3>  y_lerp, x_lerp;
    data_t a, b, c, d;

    bus64_t tmp1, tmp2;
    int last_loaded_y = 3; // (4-1)

    if (upsample == 1){
        // Prime the 4 rolling rows
        rd_Lines1a: for (int i = 0; i < 4; i++) {
            rd_Lines1b: for (int j = 0; j < map_size*channel; j++){
#pragma HLS LOOP_TRIPCOUNT max=128
#pragma HLS PIPELINE II=1
                tmp1 = strm_in.read();
                mapa_inC2D[i][j] = tmp1.data;
            }
        }

        // For each output row (2x height)
        for (ap_uint<9> oh = 0; oh < map_size*2; ++oh) {
#pragma HLS LOOP_TRIPCOUNT max=256

            if (oh == 0) { iy0 = 0; iy1 = 0; }
            else { iy0 = (oh - 1) >> 1; iy1 = iy0 + 1; }
            if (iy1 >= map_size) iy1 = map_size - 1;

            y_lerp = ((oh & 1) == 0) ? 3 : 1; // 4x fixed-point weights: 3/1

            // Load the next needed input line into the rolling buffer (before we pipeline
                the inner loop)
            if (iy1 > last_loaded_y) {
                last_loaded_y++;
                for (int j = 0; j < map_size * channel; j++) {
#pragma HLS PIPELINE II=1
                    tmp1 = strm_in.read();
                    mapa_inC2D[last_loaded_y & 3][j] = tmp1.data;
                }
            }

            // For each output col (2x width)
```

```
            for (ap_uint<9> ow = 0; ow < map_size*2; ++ow) {
#pragma HLS LOOP_TRIPCOUNT max=256

                if (ow == 0) { ix0 = 0; ix1 = 0; }
                else { ix0 = (ow - 1) >> 1; ix1 = ix0 + 1; }
                if (ix1 >= map_size) ix1 = map_size - 1;

                x_lerp = ((ow & 1) == 0) ? 3 : 1;

                // Pipeline the writer loop: 1 output word per cycle
                upsample_ch: for (int ch = 0; ch < 3; ++ch) {
#pragma HLS LOOP_TRIPCOUNT max=3
#pragma HLS PIPELINE II=1
                    if (ch >= (int)channel) continue;

                    // Cache the four 64-bit neighbors ONCE (avoid multi-reads per lane)
                    ap_uint<64> wa0 = mapa_inC2D[iy0 & 3][ix0 * channel + ch];
                    ap_uint<64> wb0 = mapa_inC2D[iy0 & 3][ix1 * channel + ch];
                    ap_uint<64> wa1 = mapa_inC2D[iy1 & 3][ix0 * channel + ch];
                    ap_uint<64> wb1 = mapa_inC2D[iy1 & 3][ix1 * channel + ch];

                    // Compute 8 lanes fully in parallel and pack to 64b
                    ap_uint<64> out_buf = 0;

                    interp_lanes: for (ap_uint<4> lane = 0; lane < 8; lane++) {
#pragma HLS UNROLL
                        data_t aa = (data_t)wa0.range(lane*8+7, lane*8);
                        data_t bb = (data_t)wb0.range(lane*8+7, lane*8);
                        data_t cc = (data_t)wa1.range(lane*8+7, lane*8);
                        data_t dd = (data_t)wb1.range(lane*8+7, lane*8);

                        // Bilinear interpolation (Q4 weights: 4-x, x)
                        ap_int<12> top    = aa * (4 - x_lerp) + bb * x_lerp;
                        ap_int<12> bottom = cc * (4 - x_lerp) + dd * x_lerp;
                        ap_int<12> value  = top * (4 - y_lerp) + bottom * y_lerp;
                        value = value >> 4;

                        // Saturate to int8
                        ap_int<9> v9 = value;
                        data_t v8 = (v9 > 127) ? (data_t)127 : (v9 < -128) ? (data_t)-128 : (
                            data_t)v9;

                        out_buf.range(lane*8+7, lane*8) = v8.range(7, 0);
                    }

                    fifo_out.write(out_buf);
                } // ch
            } // ow
        } // oh
    }
    else {
        int line_read = map_size*channel;
        if (firstLayer == 1) line_read = ((256*3)/8);

        rd_all_1: for (ap_int<10> i = 0; i < map_size; i++) {
            #pragma HLS LOOP_TRIPCOUNT max=256
            rd_all_2: for (ap_int<10> j = 0; j < line_read; j++){
                #pragma HLS LOOP_TRIPCOUNT max=256
                #pragma HLS PIPELINE II=1

                tmp1 = strm_in.read();

                // Skip connection is either on for whole frame or off; no per-iter hazard
```

```
                const bool perform_skip = (skipCon == 1) && (firstLayer == 0);
                if (perform_skip) {
                    tmp2 = skip_con_in.read();

                skip_add: for (ap_uint<4> c = 0; c < 8; c++) {
                        #pragma HLS UNROLL
                        data_t v1 = (data_t) tmp1.data.range(c*8+7, c*8);
                        data_t v2 = (data_t) tmp2.data.range(c*8+7, c*8);

                        v1 = safe_shift_ap(v1, scaleSKIP[layer_ID][1]); // input
                        v2 = safe_shift_ap(v2, scaleSKIP[layer_ID][0]); // skip

                        ap_int<9> sum = (ap_int<9>)v1 + (ap_int<9>)v2;
                        ap_int<8> sat = (sum > 127) ? (ap_int<8>)127 : (sum < -128) ? (ap_int
                            <8>)-128 : (ap_int<8>)sum;

                        tmp1.data.range(c*8+7, c*8) = sat.range(7, 0);
                    }
                }

                fifo_out.write(tmp1.data);
            }
        }
    }
}




void HW_readData(
    hls::stream<ap_uint<64>> &fifo_in, //[H][W][C]
    hls::stream<ap_uint<64>> &fifo_out,
    ap_uint<32> config
) {

    ap_uint<9> map_size = config.range(8,0);        // input dimensions
    ap_uint<2> channel = config.range(15,14);       // channel
    ap_uint<1> upsample = config.range(18,18);
    ap_uint<1> firstLayer = config.range(19,19);


    ap_uint<9> map_size_upsample = map_size + (map_size * upsample);

    ap_uint<64> auxBuf;
    ap_uint<64> zeros = 0;
    ap_uint<64> tmp;

    if (firstLayer == 1){ // For Stem
        pad1: for (ap_uint<9> i = 0; i < 258; i++)  // send zeros for padding
            fifo_out.write(zeros);

        wr_lines: for (ap_uint<9> j = 0; j < 256; j++){ // Y
            fifo_out.write(zeros); // left padding

            // scratch buffer for the 8 output 64b words produced per k
            ap_uint<64> out_words[8];
            //#pragma HLS ARRAY_PARTITION variable=out_words complete dim=1

            for (ap_uint<6> k = 0; k < 32; k++) {
                // ---- Pack phase (no stream writes here) ----
                ap_uint<64> tmp0 = fifo_in.read();
                ap_uint<64> tmp1 = fifo_in.read();
```

```
                ap_uint<64> tmp2 = fifo_in.read();

                ap_uint<64> auxBuf = 0;

                // fill out_words[0..7] exactly like your current sequence of writes:
                auxBuf.range(63,0) = ((ap_uint<40>)0, tmp0.range(23,0));
                out_words[0] = auxBuf;
                auxBuf.range(63,0) = ((ap_uint<40>)0, tmp0.range(47,24));
                out_words[1] = auxBuf;
                auxBuf.range(15,0) = tmp0.range(63,48);
                auxBuf.range(63,16) = ((ap_uint<40>)0, tmp1.range(7,0));
                out_words[2] = auxBuf;
                auxBuf.range(63,0) = ((ap_uint<40>)0, tmp1.range(31,8));
                out_words[3] = auxBuf;
                auxBuf.range(63,0) = ((ap_uint<40>)0, tmp1.range(55,32));
                out_words[4] = auxBuf;
                auxBuf.range(7,0) = tmp1.range(63,56);
                auxBuf.range(63,8) = ((ap_uint<40>)0, tmp2.range(15,0));
                out_words[5] = auxBuf;
                auxBuf.range(63,0) = ((ap_uint<40>)0, tmp2.range(39,16));
                out_words[6] = auxBuf;
                auxBuf.range(63,0) = ((ap_uint<40>)0, tmp2.range(63,40));
                out_words[7] = auxBuf;

                // ---- Emit phase: exactly one write per iteration (II=1 achievable) ----
                emit: for (int t = 0; t < 8; t++) {
                    #pragma HLS PIPELINE II=1
                    fifo_out.write(out_words[t]);
                }
            }

            fifo_out.write(zeros); // right padding
        }

        pad2: for (ap_uint<9> i = 0; i < 258; i++)  // send zeros for padding
            #pragma HLS PIPELINE II=1
            fifo_out.write(zeros);
    }
    else {
        padTop: for (ap_uint<9> i = 0; i < (map_size_upsample+2)*channel; i++)  // send zeros
             for padding
            #pragma HLS LOOP_TRIPCOUNT max=258
            fifo_out.write(zeros);

        wr_linesb: for (ap_uint<9> j = 0; j < map_size_upsample; j++){
            #pragma HLS LOOP_TRIPCOUNT max=256

            padLeft: for (ap_uint<2> k = 0; k < channel;k++){
                #pragma HLS LOOP_TRIPCOUNT max=3
                fifo_out.write(zeros);
            }

            rd_datab: for (ap_uint<9> k = 0; k < map_size_upsample*channel; k++){
                #pragma HLS LOOP_TRIPCOUNT max=256
                tmp = fifo_in.read();
                fifo_out.write(tmp.range(63,0));
            }

            padRight: for (ap_uint<2> k = 0; k < channel;k++) {
                #pragma HLS LOOP_TRIPCOUNT max=3 // 8*3 = 24
                fifo_out.write(zeros);
            }
        }
```

```
253         padBottom: for (ap_uint<9> i = 0; i < (map_size_upsample+2)*channel; i++){  // send
                zeros for padding
254             #pragma HLS LOOP_TRIPCOUNT max=258
255             fifo_out.write(zeros);
256         }
257     }
258 }
259
260
261
262 #define MAX_LINES   3             // 3 lines are enough for 3x3
263 #define KERNEL_SIZE 3
264 #define PAD         1
265 #define MAX_CH      3             // channel field is 2 bits -> [0..3] => max 3
266 #define MAX_W_PADDED 258          // worst-case width with +2 pad
267 #define BUF_COLS    (MAX_W_PADDED * MAX_CH)
268
269 void HW_conv3D(
270     hls::stream<ap_uint<64>> &fifo_in,
271     hls::stream<ap_uint<64>> &fifo_out1,
272     hls::stream<ap_uint<64>> &fifo_out2,
273     ap_uint<32> config
274 ) {
275
276     ap_uint<9> map_size = config.range(8,0);        // input dimensions
277     const unsigned int layer_ID = config.range(13,9).to_uint();       // layer identifier
278     const unsigned int channel = config.range(15,14).to_uint();       // channel
279     ap_uint<2> filters = config.range(17,16);       // channel
280     ap_uint<1> upsample = config.range(18,18);
281
282     const int scale = scale3D[layer_ID].to_int();
283
284     ap_uint<9> map_size_upsample = map_size + (map_size * upsample);
285     ap_uint<9> map_size_up_padded = (map_size_upsample+2*PAD);
286
287     ap_uint<17> maxReads =  map_size_up_padded*map_size_up_padded*channel;
288     ap_uint<17> readCounter = 0;
289
290     ap_uint<64> mapa_inC2D[MAX_LINES][BUF_COLS];
291 #pragma HLS bind_storage variable=mapa_inC2D type=RAM_2P impl=bram
292
293     // Two-dimensional accumulators: [8 filters][9 kernel positions]
294     acc_t acc1[8][KERNEL_SIZE*KERNEL_SIZE];
295     acc_t acc2[8][KERNEL_SIZE*KERNEL_SIZE];
296     #pragma HLS ARRAY_PARTITION variable=acc1 complete dim=0
297     #pragma HLS ARRAY_PARTITION variable=acc2 complete dim=0
298
299     ap_uint<64> tmp, tmp1;
300
301     rd_Lines1a: for (int i = 0; i < MAX_LINES; i++){
302         #pragma HLS LOOP_TRIPCOUNT min=3 max=3
303         rd_Lines1b: for (int j = 0; j < map_size_up_padded*channel; j++){
304             #pragma HLS LOOP_TRIPCOUNT max=258
305             #pragma HLS PIPELINE
306             tmp = fifo_in.read();
307             readCounter++;
308             mapa_inC2D[i][j] = tmp.range(63,0);
309         }
310     }
311
312     map_line: for (ap_uint<9> i = 0; i < map_size_upsample; i++) {
313     #pragma HLS LOOP_TRIPCOUNT max=256
314         map_col: for (int j = 0; j < map_size_upsample; j+=2) {
```

```
315         #pragma HLS LOOP_TRIPCOUNT max=128
316
317             ft_opt: for (int fx = 0; fx < filters; fx++) {
318                 #pragma HLS LOOP_TRIPCOUNT max=3
319
320                 // Initialize all accumulator positions to zero
321                 init_acc: for (int f = 0; f < 8; f++) {
322                     #pragma HLS UNROLL
323                     for (int k = 0; k < KERNEL_SIZE*KERNEL_SIZE; k++) {
324                         #pragma HLS UNROLL
325                         acc1[f][k] = 0;
326                         acc2[f][k] = 0;
327                     }
328                 }
329
330                 ch: for (int ch_idx = 0; ch_idx < channel; ch_idx++){
331                 #pragma HLS LOOP_TRIPCOUNT max=2
332
333                     ch_kx_ky: for (int idx = 0; idx < KERNEL_SIZE * KERNEL_SIZE; idx++) {
334                     #pragma HLS PIPELINE II=1
335                         int ki = idx / KERNEL_SIZE;
336                         int kj = idx % KERNEL_SIZE;
337
338                         int ii = i + ki;
339                         int jj  = j + kj;
340                         int jj2 = jj + 1;
341                         if (jj2 >= (int)map_size_up_padded) jj2 = (int)map_size_up_padded -
                                1;
342
343                         int map_idx1 = jj  * channel + ch_idx;
344                         int map_idx2 = jj2 * channel + ch_idx;
345
346                         ap_uint<64> v1 = mapa_inC2D[(ii%MAX_LINES)][map_idx1];
347                         ap_uint<64> v2 = mapa_inC2D[(ii%MAX_LINES)][map_idx2];
348
349                         data_mac: for (int c = 0; c < 8; c++) {
350                             #pragma HLS UNROLL
351                             data_t d1 = (data_t)v1.range(c*8+7, c*8);
352                             data_t d2 = (data_t)v2.range(c*8+7, c*8);
353
354                             filters_mac: for (int f = 0; f < 8; f++){
355                                 #pragma HLS UNROLL
356                                 data_t w1 = weights3D[layer_ID][ki][kj][fx*8+f][ch_idx*8+c];
357                                 acc_t w1_mul1, w1_mul2;
358                                 // Bind specific operations to DSP48 with latency control
359                                 #pragma HLS BIND_OP variable=w1_mul1 op=mul impl=dsp latency
                                        =3
360                                 #pragma HLS BIND_OP variable=w1_mul2 op=mul impl=dsp latency
                                        =3
361
362                                 w1_mul1 = d1 * w1;
363                                 w1_mul2 = d2 * w1;
364
365                                 // Accumulate to separate position for each kernel element
366                                 acc1[f][idx] += w1_mul1;
367                                 acc2[f][idx] += w1_mul2;
368                             }
369                         }
370                     }
371                 }
372
373                 // Reduction tree to sum all kernel positions and add bias once
374                 acc_t final_acc1[8], final_acc2[8];
```

```
                #pragma HLS ARRAY_PARTITION variable=final_acc1 complete
                #pragma HLS ARRAY_PARTITION variable=final_acc2 complete

                reduce: for (int f = 0; f < 8; f++) {
                    #pragma HLS UNROLL
                    // Start with bias
                    final_acc1[f] = bias3D[layer_ID][fx*8+f];
                    final_acc2[f] = bias3D[layer_ID][fx*8+f];

                    // Sum all kernel positions
                    for (int k = 0; k < KERNEL_SIZE*KERNEL_SIZE; k++) {
                        #pragma HLS UNROLL
                        final_acc1[f] += acc1[f][k];
                        final_acc2[f] += acc2[f][k];
                    }
                }

                data_t buf1[8], buf2[8];
                #pragma HLS ARRAY_PARTITION variable=buf1 complete
                #pragma HLS ARRAY_PARTITION variable=buf2 complete

                res: for (ap_uint<4> c = 0; c < 8; c++){
                    #pragma HLS UNROLL
                    data_t v1 = (data_t)CLAMP8_RELU_AP(final_acc1[c] >> scale);
                    data_t v2 = (data_t)CLAMP8_RELU_AP(final_acc2[c] >> scale);
                    buf1[c] = v1;
                    buf2[c] = v2;
                }

                ap_uint<64> bufout1, bufout2;
                store: for (ap_uint<4> p = 0; p < 8; p++){
                    #pragma HLS UNROLL
                    bufout1.range(p*8+7, p*8) = buf1[p].range(7,0);
                    bufout2.range(p*8+7, p*8) = buf2[p].range(7,0);
                }

                fifo_out1.write(bufout1);
                fifo_out2.write(bufout2);
            }
        }

        if(readCounter < maxReads) {
            int line = (i + MAX_LINES);
            for (int x = 0; x < map_size_up_padded*channel; x++){
                #pragma HLS LOOP_TRIPCOUNT max=258
                #pragma HLS PIPELINE II=1
                tmp1 = fifo_in.read();
                readCounter++;
                mapa_inC2D[(line%MAX_LINES)][x].range(63,0) = tmp1.range(63,0);
            }
        }
    }
}


void HW_writeData(
    hls::stream<bus64_t>      &strm_out,
    hls::stream<ap_uint<64>>  &fifo_in1,
    hls::stream<ap_uint<64>>  &fifo_in2,
    config_t                   config
){
#pragma HLS INTERFACE axis port=strm_out
#pragma HLS INLINE off
```

```
438
439     const unsigned int map_size = config.range(8,0).to_uint();        // input dimensions
440     const unsigned int filters = config.range(17,16).to_uint();       // channel
441     const unsigned int upsample = config.range(18,18).to_uint();
442
443     const unsigned int map_size_up = map_size * (1 + upsample);
444     const unsigned int half_j      = (map_size_up >> 1);
445
446     // total number of 64-bit words produced for this frame
447     const unsigned int TOTAL = filters * map_size_up * map_size_up;
448
449     unsigned emitted = 0;
450
451     bus64_t outw;
452     outw.keep = 0xFF;
453     outw.strb = 0xFF;
454     outw.last = 0;
455
456 RowLoop:
457     for (unsigned i = 0; i < map_size_up; ++i) {
458     #pragma HLS LOOP_TRIPCOUNT min=1 max=256
459     ColPairLoop:
460         for (unsigned j = 0; j < half_j; ++j) {
461         #pragma HLS LOOP_TRIPCOUNT min=1 max=128
462
463             // Emit the first column group (fifo_in1): filters words
464         Filt1Loop:
465             for (unsigned k = 0; k < filters; ++k) {
466             #pragma HLS PIPELINE II=1
467                 ap_uint<64> w = fifo_in1.read();
468                 outw.data = w;
469                 // TLAST only on very last word of the frame
470                 outw.last = ((emitted + 1) == TOTAL);
471                 strm_out.write(outw);
472                 emitted++;
473             }
474
475             // Emit the second column group (fifo_in2): filters words
476         Filt2Loop:
477             for (unsigned k = 0; k < filters; ++k) {
478             #pragma HLS PIPELINE II=1
479                 ap_uint<64> w = fifo_in2.read();
480                 outw.data = w;
481                 outw.last = ((emitted + 1) == TOTAL);
482                 strm_out.write(outw);
483                 emitted++;
484             }
485         }
486     }
487 }
488
489
490
491 void HW_layerC3D(
492     hls::stream<bus64_t> &strm_in,
493     hls::stream<bus64_t> &skip_in,
494     hls::stream<bus64_t> &strm_out,
495     config_t config
496 ){
497 #pragma HLS interface axis port=strm_in
498 #pragma HLS interface axis port=skip_in
499 #pragma HLS interface axis port=strm_out
500 #pragma HLS INTERFACE s_axilite port=return bundle=AXILite
```

```
#pragma HLS INTERFACE s_axilite port=config bundle=AXILite
#pragma HLS DATAFLOW

static hls::stream<ap_uint<64>> fifoA("fifoA");
static hls::stream<ap_uint<64>> fifoB("fifoB");
static hls::stream<ap_uint<64>> fifo1("fifo1");
static hls::stream<ap_uint<64>> fifo2("fifo2");

#pragma HLS STREAM variable=fifoA depth=2048
#pragma HLS STREAM variable=fifoB depth=2064
#pragma HLS STREAM variable=fifo1 depth=2048
#pragma HLS STREAM variable=fifo2 depth=2048

    HW_upsample(strm_in, skip_in, fifoA, config);
    HW_readData(fifoA, fifoB, config);
    HW_conv3D(fifoB, fifo1, fifo2, config);
    HW_writeData(strm_out, fifo1, fifo2, config);
}
```

Listing D.3: Convolution 3D Core HLS Code

E Mobile-CMUNeXt Architecture

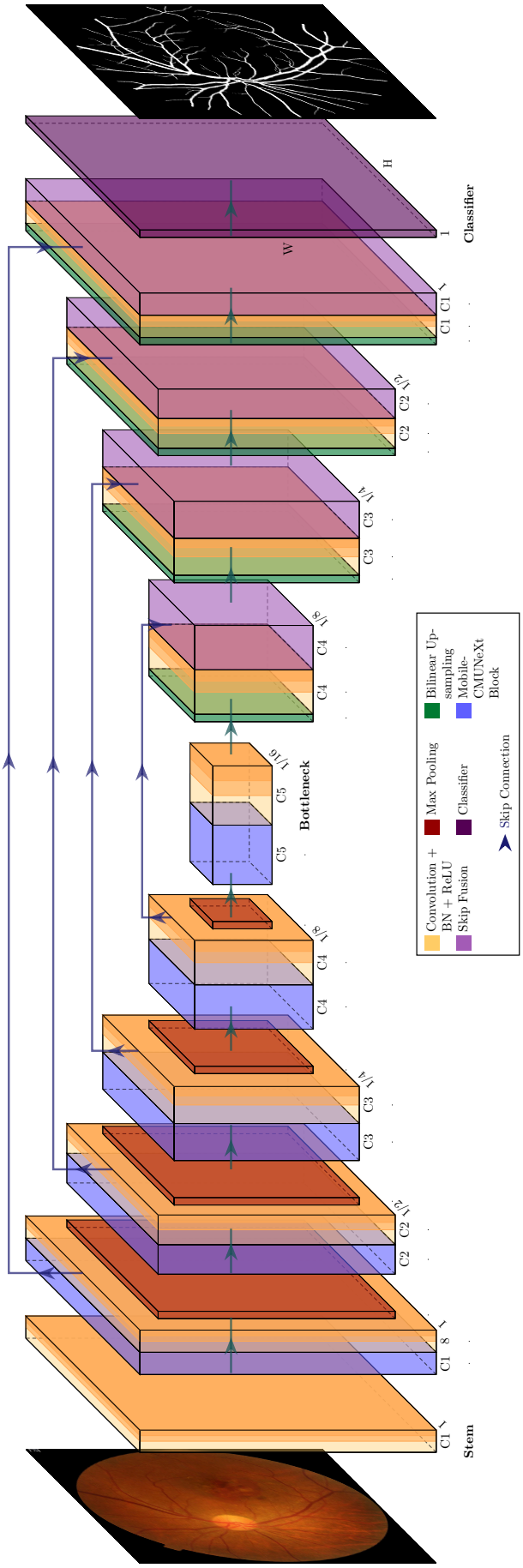


Figure E.1: Full Mobile-CMUNeXt architecture diagram with annotated images.

Semantic Segmentation of Medical Images for Fast Diagnosis

This document was created using the (pdf)LaTeX processor.