Efficient Processing of Convolutional Neural Networks on the Edge: A Hybrid Approach Using Hardware Acceleration and Dual-Teacher Compression

Azzam Alhussain
University of Central Florida

EFFICIENT PROCESSING OF CONVOLUTIONAL NEURAL NETWORKS ON THE EDGE: A HYBRID APPROACH USING HARDWARE ACCELERATION AND DUAL-TEACHERS COMPRESSION

by

AZZAM ALHUSSAIN
M.S. Electrical Engineering, University of Dayton, 2019
B.S. Instrumentation & Control Eng. Tech, Jubail Industrial College, 2014

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2024

Major Professor: Mingjie Lin

ABSTRACT

Accelerating Deep Learning frameworks, particularly Convolutional Neural Networks (CNNs) for computer vision applications on the edge require the development of specialized computing solutions capable of maintaining high accuracy and performing real-time inference. This research is motivated by the open-source hardware design frameworks such as FINN and HLS4ML, and its primary focus is on hardware acceleration, model compression, and efficient implementation of computer vision CNNs algorithms on the AMD SoC-FPGAs using High-Level Synthesis (HLS) to enhance on-chip data structures and optimize the resource utilization, thereby improving the throughput/watt of FPGA-based AI accelerators compared to a traditional fixed-logic chips, like CPU and GPU implementations, as well as other edge accelerators.

Furthermore, this dissertation proposed unique solutions to hardware and software co-design approaches for accelerating deep learning algorithms, specifically focusing on CNNs, Generative Adversarial Networks (GANs), and Human Action Recognition (HAR) models implemented on AMD ZYNQ SoC-FPGAs edge chips. Our methodologies leverage advanced techniques such as quantization, knowledge distillation, loop tiling transformation, and dataflow modeling to optimize the performance, accuracy, and resource utilization of these models.

Additionally, we introduced a novel CNN compression technique named "Two-Teachers Net," which utilizes PyTorch FX-graph mode to train an 8-bit quantized student model using dual-teachers knowledge distillation approach. It improved the accuracy of the compressed model by 1% - 2% compared to the existing solutions for edge platforms. This method can be applied to any CNN model and dataset for image classification and seamlessly integrated into existing AI hardware and software optimization toolchains, including Vitis-AI, OpenVINO, TensorRT, and ONNX, without architectural adjustments. This provides a scalable solution for deploying high-accuracy CNNs on low-power edge devices across various applications, such as autonomous vehicles, surveillance systems, robotics, healthcare, and smart cities.

In summary, this dissertation aims to push the state-of-the-art in efficient compression and resource scheduling of CNNs on resource-constrained platforms through FPGA hardware acceleration. It provides a scalable solution for deploying high-accuracy CNNs on low-power edge devices for various industries applications, including autonomous vehicles, surveillance systems, robotics, healthcare, and smart cities.

To my parents, wife, brothers, and sisters for their endless support and encouragement throughout this journey. I could not have done this without you.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Mingjie Lin, for his invaluable guidance, support, and feedback throughout my Ph.D. studies. He has been an excellent mentor for me, and I am honored to have worked with him.

I am also grateful to the members of my dissertation committee, Dr. Yaser Fallah, Dr. Rickard Ewetz, Dr. Jun Wang, and Dr. Hatem Abou-Senna for their constructive comments and suggestions that helped me improve the quality of my dissertation.

My research would not have been possible without the generous funding from the Royal Commission for Jubail & Yanbu and the University of Central Florida Graduate School, which enabled me to pursue my Ph.D. studies in the United States.

I would like also to express my appreciation to Dr. Song Han at MIT, whose research, knowledge, and open-source projects on the subject refine my thinking and brought my work to a higher level.

Finally, I must express my profound gratitude to my parents, wife, and friends for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE: INTRODUCTION

Convolutional Neural Networks (CNNs) have upscaled the field of computer vision in many applications, such as image classification, object detection, and instance segmentation [1]. CNNs are Deep Neural Networks (DNNs) designed to handle structure input data in grid representations of pixel values by leveraging the spatial and temporal dependencies [2]. As the demand for Artificial Intelligence (AI) and autonomous systems increases, there is a growing interest in utilizing DNNs and CNNs on edge devices, such as smartphones, drones, and IoT sensors. This is done to enable real-time data and low-latency inference [3]. For instance, EfficientNetV2 [4] is a state-of-the-art (SOTA) lightweight CNN architecture. It leverages a unique compound scaling method to systematically scale up the depth, width, and resolution of CNNs. This scaling approach effectively reduces the model size, computing cost, and number of parameters. The model achieved superior performance with an accuracy of top-1 87.3% for the ImageNet [5] classification challenge. Fig. 1 shown a simple design for CNN architecture used in most of this dissertation.

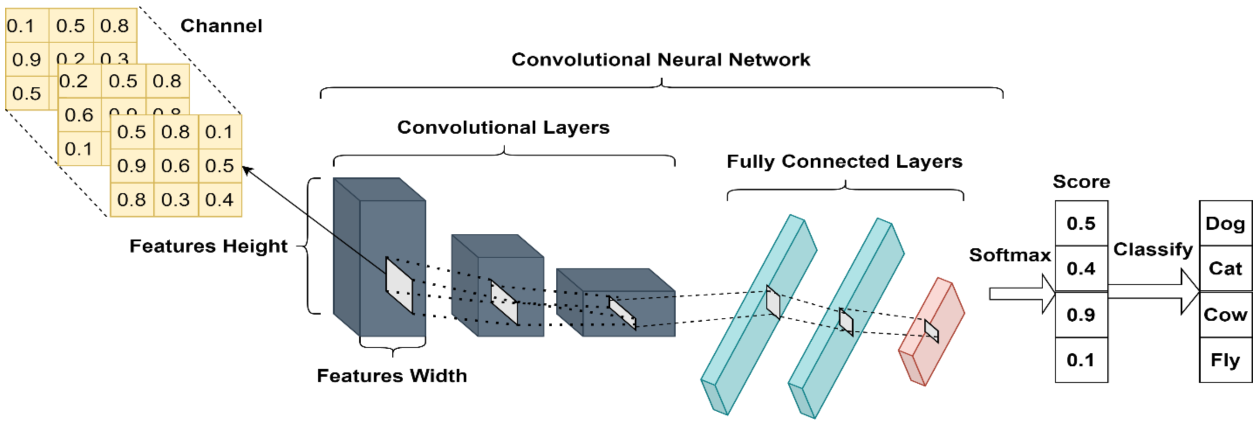

Figure 1. Simple CNN architecture design.

However, the computational requirements of CNNs pose considerable obstacles to their implementation on resource-constrained platforms [6]. For example, the EfficientNet-B7 model [7] consists of 66 million parameters and requires 37 billion floating-point operations (FLOPs) to process a single image. This translates to a memory requirement of 256 MB and a computational demand of 74 Giga Floating-Point Operations per Second (GFLOPS) [4]. These requirements exceed the capabilities of most edge devices, such as smartphones, IoT sensors, and low-power processors (e.g., ARM Cortex-A series) [8]. Additionally, real-time inference on the edge often demands a high processing throughput of 24 frames per second (fps) for video applications. This further increases the computing load [9].

Nevertheless, CNNs demonstrated superior performance and efficiency compared to Vision Transformers (ViTs) on smaller datasets, making them a good choice for edge deployment. On the CIFAR-10 dataset, MobileNetV3 [10] achieved a state-of-the-art accuracy of 98.22% with only 2.3M parameters and 56.6M multiply–accumulate (MACs), while the ViT [11] requires 5.7M parameters and 246.8M MACs to achieve a lower accuracy of 97.12%. Similarly, on the CIFAR-100 dataset, ShuffleNetV2 [12] achieves an accuracy of 95.45% with 3.5M parameters and 134.2M MACs, outperforming the ViT, which achieved an accuracy of 94.55%. These results demonstrate the superior efficiency and accuracy of CNNs on small datasets, making them the choice for edge deployment, where computational resources are limited. Fig. 2 below shown the most recent graph for a CNN model trained on CIFAR-10 dataset, which outperformed the ViT in terms of accuracy.

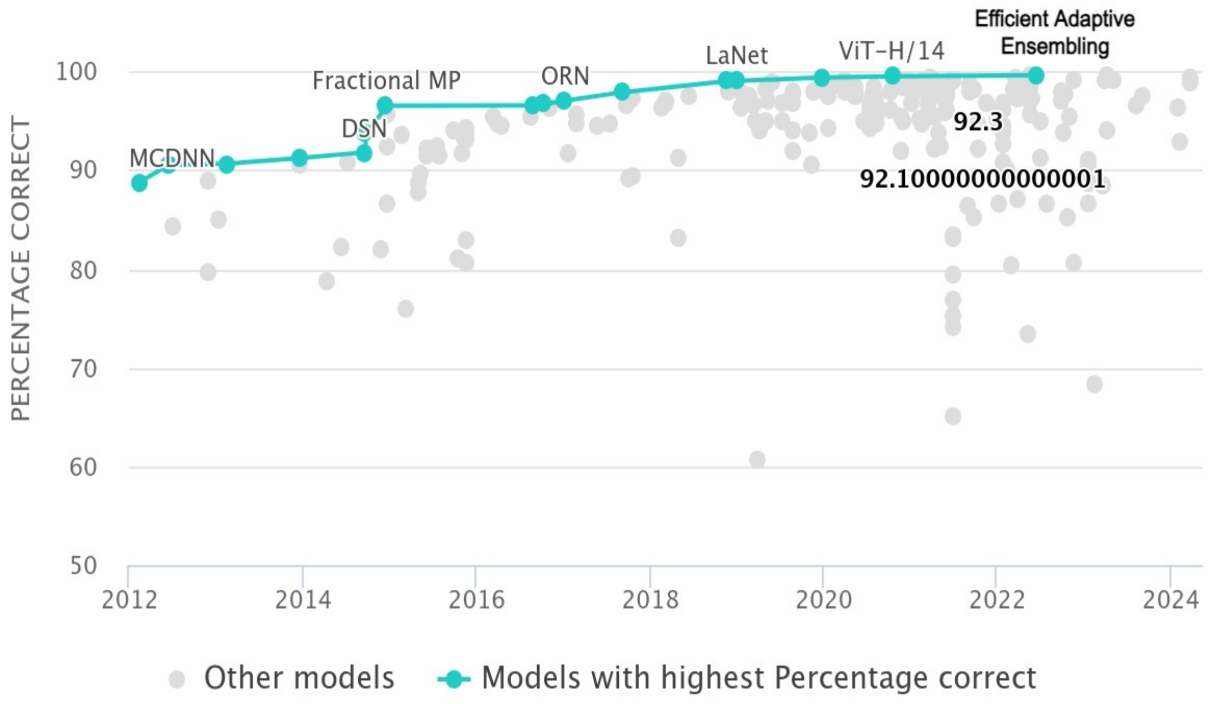

Figure 2. The Efficient Adaptive Ensembling CNN model outperformed ViT in terms of accuracy on CIFAR-10 dataset [13].

Moreover, CNNs have also been shown to be more efficient in terms of training time and memory requirements. For example, the training time for MobileNetV3 on CIFAR-10 is approximately 10 hours on a single NVIDIA V100 GPU, while the training time for the ViT is approximately 24 hours on the same hardware. Similarly, the memory requirements for CNNs are significantly lower than those for ViTs, with MobileNetV3 requiring approximately 1.2 GB of memory to train on CIFAR-10, compared to 3.5 GB for the ViT.

However, several methods have been suggested to optimize DNNs for efficient deployment on edge devices while still achieving high accuracy and real-time performance. These techniques include model compression [14] and hardware acceleration [15].

Motivation

Fig. 3 illustrates the design philosophy of this thesis and shows an end-to-end algorithm optimization and hardware co-design to make DNN inference more efficient on dedicated SoC-FPGA accelerators compared to GPU and CPU.

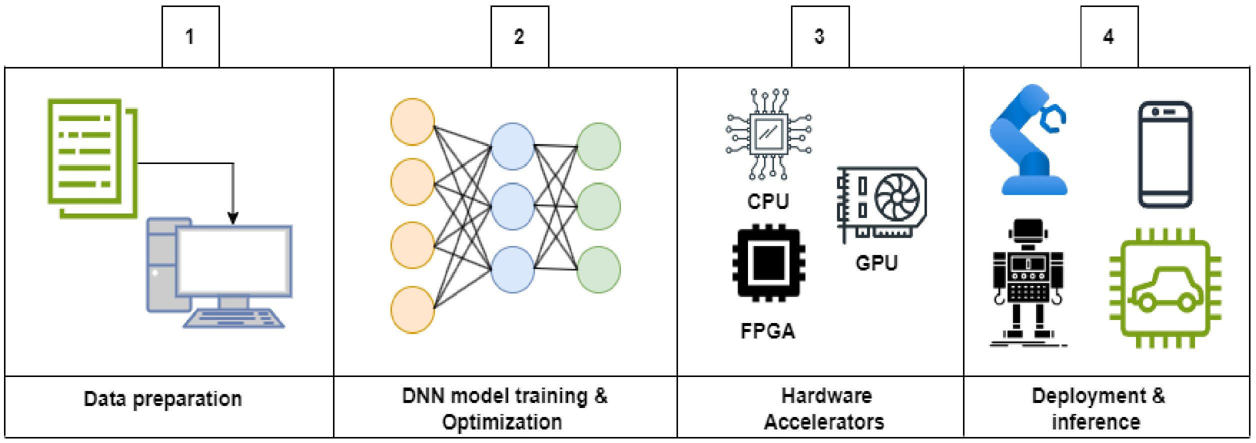

Figure 3. A complete HW/SW flow co-design in sequence (1-4) for deep learning that aims to address algorithms compression techniques for high accuracy and fast inference.

Fig. 4 is shown some model compression techniques that helps reduce the model size and computational complexity of DNN architectures while maintaining high accuracy. Pruning is one technique that involves removing redundant or less essential weights, filters, or channels from the model [16]. Cai et al. [17] proposed a channel pruning method for CNNs that achieved a 2× reduction in model size and a 3× reduction in computational complexity for the EfficientNet-B0 model on the ImageNet dataset, with only a 0.3% drop in top-1 accuracy. Quantization is another effective technique that reduces the precision of weights and activations. It usually converts them from 32-bit floating-point to 8-bit or even 1-bit integers [18]. Nagel et al. [19] introduced a novel quantization scheme called AdaRound. This method resulted in a 4× reduction in model size and a 2× speedup in inference time for the MobileNetV2 model on the ImageNet dataset, with only a 0.3% drop in top-1 accuracy. Knowledge Distillation is also another helpful technique that transfers knowledge from a large, complex model (teacher) to a more minor, more straightforward (student) model [20]. Yun et al. [21] proposed a novel knowledge distillation method called RKD, which achieved a 1.2% higher top-1 accuracy than the original EfficientNet-B0 model on the ImageNet dataset while being 2× smaller and 1.5× faster. These model compression techniques can be combined to achieve even higher efficiency gains for edge inference [22].

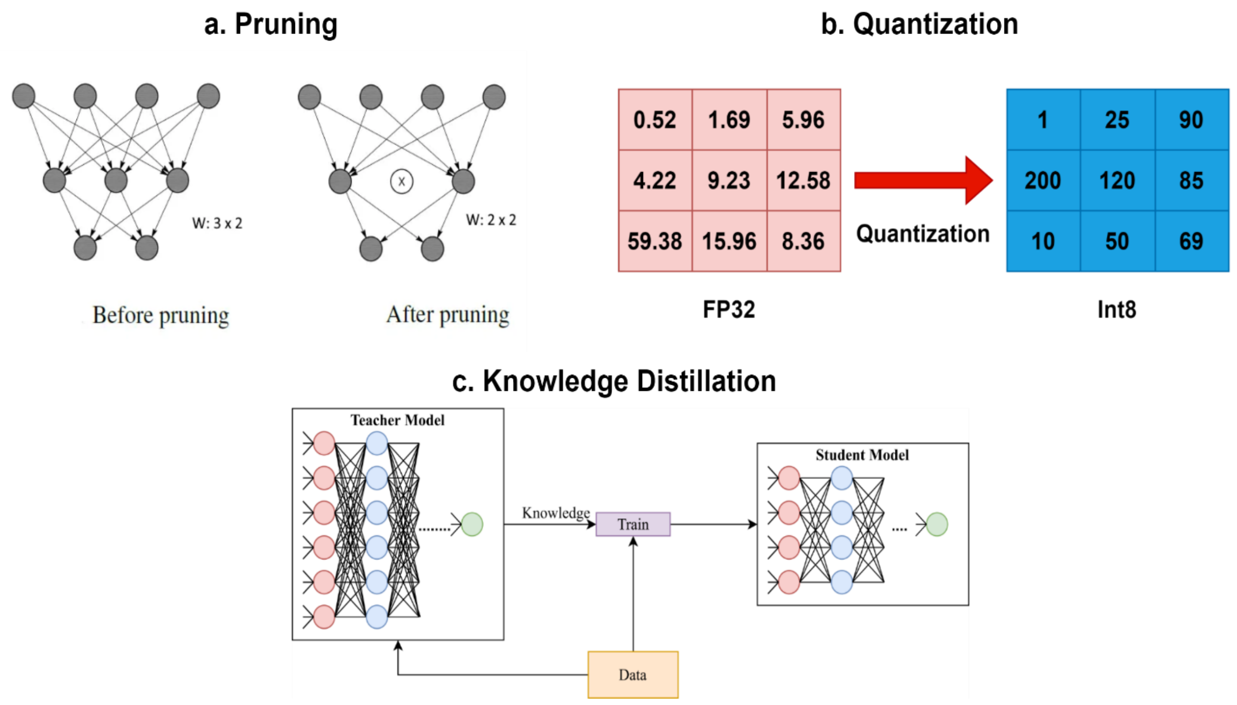

Figure 4. a. Pruning, b. Quantization, and c. Knowledge Distillation

On the other hand, researchers have investigated many hardware platforms, such as Graphics Processing Units (GPUs), Central Processing Units (CPUs), Application-Specific Integrated Circuits (ASICs), Tensor Processing Units (TPUs), and Field-Programmable Gate

Arrays (FPGAs), to speed up DNN inference on edge devices [23]. Table 1 below is shown a comprehensive comparison among the mentioned hardware.

Table 1. A comprehensive comparison among the different hardware used in AI computing.

	GPU	CPU	ASIC	TPU	FPGA
Purpose	General-purpose computing	General-purpose computing	Specialized computing for specific tasks	AI processing, deep learning; and data center	Reconfigurable computing for various tasks
Architecture	Many-core, parallel processing	Few-core, serial processing	Custom-designed for specific tasks, fixed-function	Custom-designed for AI, systolic array	Reconfigurable logic elements, LUTs (Look-Up Tables)
Performance	High, 10-100 GFLOPS	High, 10-100 GFLOPS	High, 1-10 Gbps	High, 10-100 TFLOPS	High, 1-10 GFLOPS
Power consumption	High, 50W-500W	Medium, 5W-100W	Low, 0.1W-10W	Low, 5W-50W	Low, 1W-25W
Cost	Medium to High, \$500-\$10,000	Low to Medium, \$50-\$1,000	High, \$1,000-\$50,000	High, \$1,000-\$50,000	Low to Medium, \$250-\$10,000
Flexibility	Multi-GPUs support	Multi-cores support	Fixed function	Fixed function	Reconfigurable computing; multi-FPGAs support
Scalability	High, multi-GPU support; 10-100 GPUs	Medium, multi-core support; 2-16 cores	Low, fixed-function; 1-10 ASICs	Low, fixed-function; 1-10 TPUs	High, multi-FPGAs support; 10-100 FPGAs

GPUs are widely used for training DNNs and offer high computational throughput and parallelism. This makes them well-suited for accelerating inference on edge devices with relatively high-power consumption [24]. The NVIDIA Jetson Xavier NX is an example of a popular edge GPU that can deliver up to 21 Tera Operations Per Second (TOPS) for INT8 inference. This enables real-time performance for complex models like YOLOv5 [25]. Conversely, CPUs are more common and offer flexibility, but their performance is limited compared to dedicated accelerators [26]. ASICs, such as Google's Edge TPU and Intel's Movidius Myriad X, are highly optimized for specific DNN models and offer the best performance and energy efficiency. However, they lack flexibility and have high development costs [27], [28]. Due to their reconfigurable fabric and low power consumption, FPGAs have become famous for edge inference because of their flexibility and energy efficiency [29]. As an illustration, the AMD SoC-FPGA ZYNQ UltraScale+ MPSoC can achieve up to 38 TOPS/W for INT8 inference. This ZYNQ architecture [30] features a unique Programmable Logic to Processing System (PL-PS) interface that utilizes the Advanced eXtensible Interface (AXI) protocol. This interface enables high-speed, low-latency data transfer between the FPGA fabric (PL) and the CPU (PS), which typically consists of ARM Cortex-A processors. Nevertheless, FPGAs present a more challenging learning process and demand specialized programming expertise (e.g., VHDL) compared to alternative platforms [31].

However, developing custom hardware accelerators using System-on-Chip Field-Programmable Gate Array (SoC-FPGA) with High-Level Synthesis (HLS) for CNN-based computer vision algorithms presents several challenges and difficulties. The CNN algorithms require significant computational resources and memory bandwidth [32]. Mapping these algorithms onto the FPGA fabric efficiently while optimizing for performance, resource utilization, and power consumption is a non-trivial task [33]. The HLS design methodology introduces additional complexity, such as managing data dependencies, optimizing memory access patterns, and ensuring efficient HW-SW partitioning [34] as shown in the sequence design in Fig. 5.

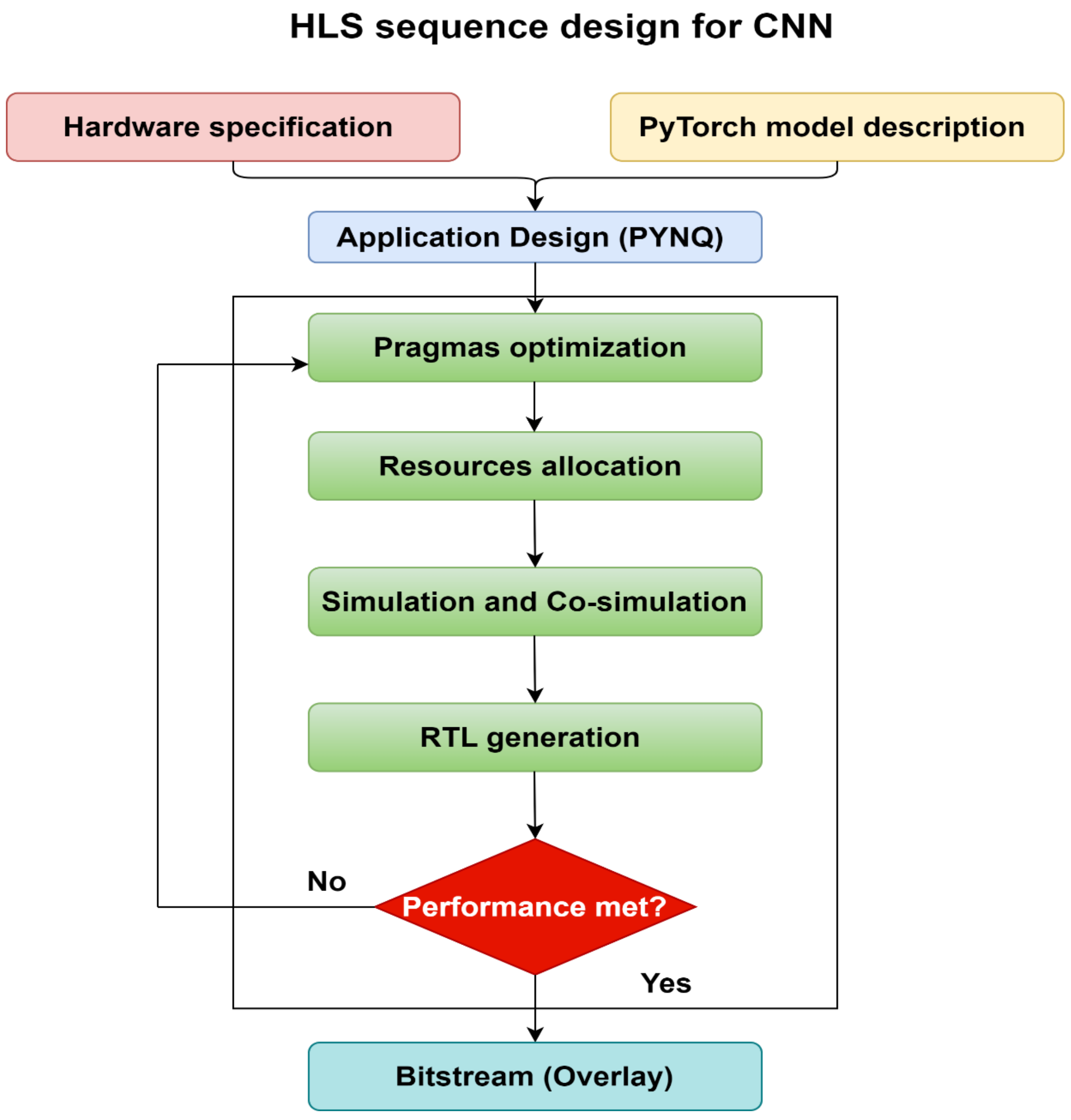

Figure 5. The HW/SW co-design required for CNN using Vivado HLS and PYNQ framework [35].

Additionally, implementing techniques like on-chip buffers, custom memory hierarchies, and efficient data reuse requires careful design considerations and trade-offs between resource utilization and performance [36]. Besides this, achieving real-time performance needs careful optimization techniques, such as loop unrolling, pipelining, and dataflow analysis, which can be challenging to implement effectively in an HLS context [37]. Furthermore, the limited resources available on FPGAs, such as logic elements, memory blocks RAM (BRAM), and DSP slices, impose limitations on the size of the CNN models that can be accelerated [38]. Despite these challenges, researchers and industry practitioners are actively developing efficient SoC-FPGA generic design methodologies and tools to accelerate CNN-based computer vision applications [39].

Objectives

DNNs and CNNs have revolutionized the field of computer vision, enabling unprecedented levels of accuracy in tasks such as image classification, object detection, and semantic segmentation [2]. However, the computational complexity and memory requirements of these networks pose significant challenges when deploying them on resource-constrained edge devices for real-time inference [6]. The limited processing power, memory bandwidth, and energy efficiency of edge devices hinder the widespread adoption of DNNs and CNNs in applications that demand low latency, high throughput, and energy efficiency [40]. To address these challenges, model compression and hardware acceleration techniques have emerged as a promising solution.

First, techniques such as pruning, quantization, and knowledge distillation have been proposed to compress CNN models [41]. Pruning involves removing redundant or less essential weights and connections, while quantization reduces the precision of weights and activations. Knowledge Distillation further up-scale the accuracy for the quantized student weights by mimic the larger teacher model to frequently occurring values. However, the effectiveness of these compression techniques varies depending on the CNN architecture and the target hardware platform [42].

Second, among various hardware platforms, AMD SoC-FPGAs with ZYNQ architecture shown in Fig. 6 have gained significant attention due to their flexibility, reconfigurability, and high performance. SoC-FPGAs combine the programmability of software with the parallel processing capabilities of the hardware, making them suitable for accelerating DNNs and CNNs [43]. The programmable logic (PL) fabric of SoC-FPGAs enables the implementation of custom hardware accelerators, while the integrated processing system (PS) ARM allows for flexible software control and data management.

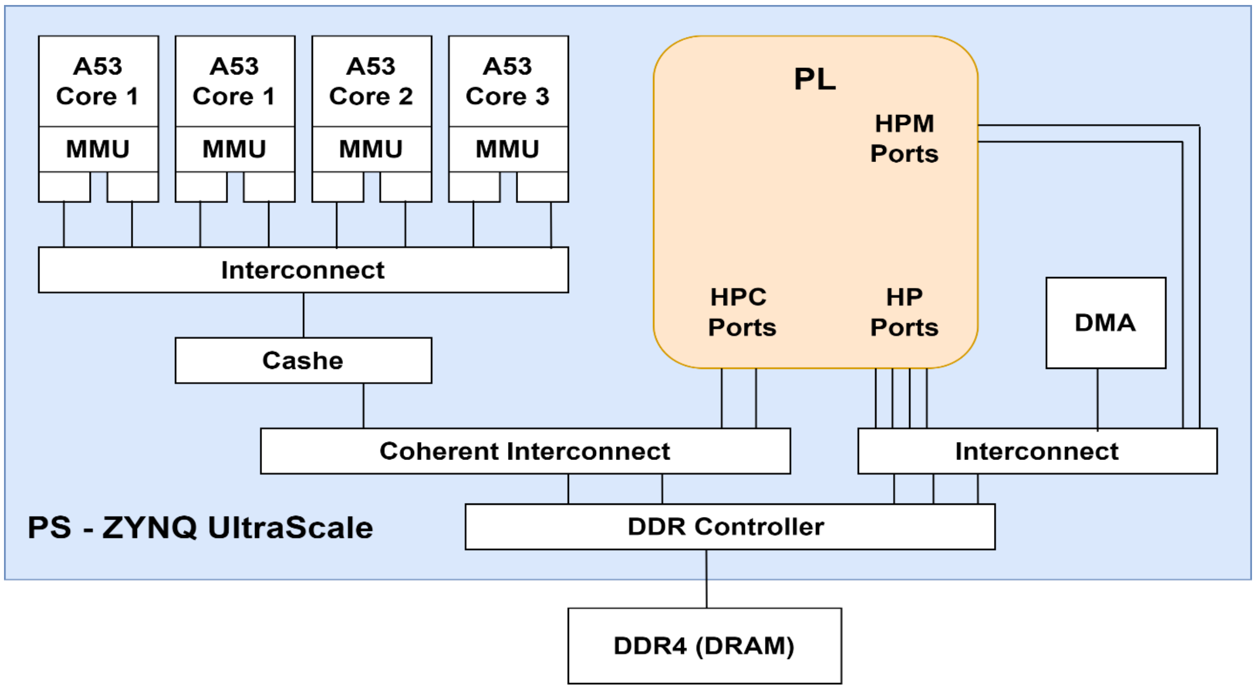

Figure 6. SoC-FPGA ZYNQ [44] architecture used in the whole design of this dissertation. The PS part has four core ARM processing unit while the PL part has high-performance ports to send/receive data. And a DDR4 which is the external memory.

However, the limited on-chip memory and bandwidth of FPGAs require efficient data reuse and memory access patterns [45]. The dataflow architecture and parallelization strategies need to be optimized based on the compressed model structure and the available hardware resources. Additionally, the use of HLS tools has emerged as a promising approach to bridge the gap between software and hardware design, enabling faster development cycles and improved productivity.

To demonstrate the effectiveness of CNN compression and hardware acceleration on AMD SoC-FPGAs, several benchmarking studies have been conducted. For example, Sun et al. [46] presented a mixed-precision quantization scheme for CNN acceleration on an AMD Zynq SoC-FPGA in 2022. They utilized 8-bit and 4-bit quantization for different layers of the different networks, using intra-layer, mixed-precision quantization. Each layer assigns 4-bit precision to 95% of filters and 8-bit precision to 5% of filters, along with 5-bit activations, and employs an optimized accelerator architecture with techniques like DSP packing, weight reordering, data packing, and a comprehensive resource allocation model, achieving accuracy comparable to 32-bit designs and over 500 fps throughput comparable to 4-bit designs on AMD ZCU102 SoC-FPGAs for ResNet-50 and MobileNetV2 models. Similarly, Shao et al. [47] proposed a friendly CNN compression method that utilized maps by transforming the stored data into frequency domain using a hardware-implemented 8×8 discrete cosine transform. They co-designed the compression algorithm and hardware architecture to maximize the utilization of FPGA resources. Their implementation on an AMD Zynq SoC-FPGA achieved a compression ratio of 30% and a throughput of over 400 GOPS for a compressed VGG-16 and ResNet-50 model, outperforming previous works in terms of both compression ratio and acceleration performance.

Despite the potential of SoC-FPGAs and HLS for accelerating DNNs and CNNs, several challenges still need to be addressed. These include the efficient partitioning of workloads between hardware and software components, the optimization of data movement and memory access patterns on BRAM, and the exploration of different parallelization strategies such as loop tiling, loop unrolling, and double buffering. This research aims to address these challenges by investigating novel hardware acceleration techniques for CNNs on AMD SoC-FPGAs with ZYNQ architecture. The focus will be on developing efficient HW-SW co-design methodologies, exploring HLS-based acceleration techniques, and optimizing real-time inference of computer vision CNN algorithms for AMD Fast Inference Neural Network (FINN) [48] and HLS4ML [49]. The research will also contribute to the advancement of AI frameworks by enabling a new compression method that maintained high accuracy for CNN image classification models on resource-constrained devices, opening up new possibilities for intelligent and autonomous systems applications. Lastly, the trade-offs between compression ratio, accuracy, and hardware performance need to be carefully analyzed and balanced.

The purpose of this study is to eliminate the challenges encountered during DNNs model compression and develop a CNN-based computer vision HW-SWs co-design across various AI frameworks for deployment on edge AMD ZYNQ SoC-FPGAs boards using HLS for high accuracy and high performance. A brief summary of the goals is as follows:

- Develop a novel CNN compression method to reduce model size and computational complexity while preserving accuracy. Advanced approaches, including pruning, quantization, and knowledge distillation, will be explored to provide cutting-edge techniques that shrink CNN models for edge deployment [50].

- Supporting popular DNN frameworks like FINN and HLS4ML to efficiently deploy different CNN-based algorithms on edge devices. This will need HW-SW co-design methods that maximize compressed CNN model mapping onto the target hardware platform.
- Accelerating different HW-CNN architectures on AMD SoC-FPGA platforms, and benchmark their performance and inference time against traditional CPU and GPU implementations. This will accelerate the execution of compressed CNN models for low-latency and energy-efficient for the edge.

The proposed methods will be evaluated using relevant performance metrics, such as inference latency, throughput, energy efficiency, and resource utilization, to demonstrate their effectiveness and practicality in real-world computer vision applications.

Contributions

This dissertation explores the co-design of algorithms compression and hardware reconfigurability for Edge AI CNN-based computer vision applications on AMD SoC-FPGA platforms [32]. The primary focus is on developing high-accuracy and high-performance solutions that enable real-time inference using the open-source FINN [51] and HLS4ML [52] framework while considering the limitations of computational resources, memory bandwidth, and power consumption for edge FPGAs.

However, edge FPGAs often have limited resources such as memory bandwidth and available resources (LUT, DSP, and FF), which makes it challenging to run computationally intensive CNN models in real-time. For instance, state-of-the-art CNN models like YOLOv8x [53] and EfficientNet-B7 [7] require billions of floating-point operations (FLOPs) and millions of parameters, making them difficult to deploy on resource-constrained edge devices. Therefore, there is a pressing need for efficient and optimized hardware-software solutions that can accelerate CNN inference on edge devices while meeting the strict latency, throughput, and energy efficiency requirements.

The proposed research addresses this need by focusing on the development of a comprehensive HW-SW co-design optimization framework that leverages the capabilities of AMD SoC-FPGA with ZYNQ architecture and HLS techniques. The ZYNQ architecture, which combines a powerful ARM processor with PL fabric, offers a unique opportunity to accelerate CNN inference through hardware-software partitioning and optimization. By exploring various aspects of the framework, such as partitioning and mapping of CNN layers such as layer-level, channel-level, and kernel-level partitioning, optimization of data transfer and communication using high-speed interconnects like AXI, exploring the use of Direct Access Memory (DMA) engines and data buffering techniques to minimize latency and maximize throughput, HLS design patterns and directives for efficient implementation on the PL fabric like pipelining, loop unrolling, dataflow, array partitioning, loop tiling, resource sharing, scalable hardware architecture design using reconfigurable computing techniques, and software optimization techniques for model compression like quantization, pruning, and knowledge distillation, this research aims to provide a holistic solution for accelerating CNN inference on edge devices.

The first contribution is an HLS-based scalable dataflow inference accelerator for transpose convolution in quantized Deconvolutional Generative Adversarial Networks [54] (QDCGAN), built on the FINN framework. We provided an open-source framework covering training to hardware implementation, allowing researchers to study the impact of different bit widths for weights and activations on performance, resource usage, throughput, and image quality. The proposed accelerator uses an efficient deconvolution engine enabling high parallelism for GAN-based edge FPGA computing. Various precisions, datasets, and scalability options were analyzed for low-power edge inference, with performance benchmarked against NVIDIA Jetson Nano. This work encourages the exploration of efficient implementation of super-resolution GANs on low-power FPGAs, potentially aiding the deployment of technologies like NVIDIA DLSS [55] on edge platforms. This work was published in the 2022 IEEE 56th Annual Conference on Information Sciences and Systems (CISS) [56].

Next, we proposed a scalable HLS-based architecture built on the HLS4ML framework that efficiently maps pre-trained CNN models with 16-bit fixed-point quantization onto a hardware template, achieving a high performance-to-resource utilization ratio. The design leverages loop tiling transformation and dataflow modeling to optimize convolutional and fully connected layers for on-chip vector multiplication. This allowed the accelerator IP to transfer a fixed amount of data from DRAM to BRAM, enabling efficient computations. A comparative analysis with previous developments demonstrated that the proposed method achieved 1.3x - 1.7x higher performance (230 GOP/s) operated at 200-MHz with minimum data execution time on well-known networks like AlexNet [57], VGG16 [58], and LeNet. This work was published in the 2022 IEEE International Conference on Networking, Architecture and Storage (NAS) [59].

After that, we presented an innovative approach to accelerate Human Action Recognition (HAR) on edge devices using SoC-FPGA. We developed a full-stack scalable HW/SW co-design based on an enhanced 8-bit quantized Two-Stream SimpleNet-PyTorch [60] CNN architecture. We fused convolutional, batch-norm, and ReLU operations into a single layer and utilized the Lucas-Kanade motion flow method to enhance the intellectual property (IP) parallelism and optimized on-chip computing. The design demonstrated significant improvements over prior work, achieving nearly 81% prediction accuracy with an approximate 24 FPS real-time inference throughput at 187MHz on ZCU104. This performance represents a 1.7x - 1.9x increase in throughput compared to previous research. This work was published in the 2023 IEEE 20th International Conference on Smart Communities: Improving Quality of Life using AI, Robotics, and IoT (HONET) [61].

In the last contribution, we presented a novel generic compression technique called "Two-Teachers Net," which utilizes PyTorch FX-graph mode to train an 8-bit quantized student model using knowledge distillation from two teacher models. This innovative method improves the accuracy of the compressed model by 1%-2% compared to existing solutions for edge platforms and can be seamlessly integrated into existing AI hardware and software optimization toolchains without architectural adjustments. This methodology aimed to create a generic scalable solution on top of Vitis-AI [62], OpenVINO [63], TensorRT [64], and ONNX [65] edge AI frameworks. This is done without architectural adjustments to the predefined dimension layers to prove the effectiveness of hardware-algorithm co-design through experimental results in real-time inference [66]. This work has been submitted to the 2024 IEEE 21st International Conference on Smart Communities: Improving Quality of Life using AI, Robotics, and IoT (HONET).

Dissertation Organization

This dissertation is organized as follows: Chapter 2 describes a detailed background on DNN and CNN compression approaches alongside the HLS-based accelerator for SoC-FPGA. Then, we present an edge scalable inference accelerator for the GAN algorithm built on the FINN framework in Chapter 3. In Chapter 4, we explored an efficient CNN architecture change built on the HLS4ML framework using on-chip vector multiplication for generic types of layers on the FPGA. Chapter 5 introduced an enhanced CNN-based HAR accelerator design by fusing most layers into a single layer for better data parallelism on edge. Additionally, Chapter 6 proposed a generic CNN compression technique using quantization and dual-teacher knowledge distillation to improve the accuracy by 1%-2% compared to exiting solutions and frameworks available for edge implementation. Finally, Chapter 7 concludes the dissertation and discusses the future work.

CHAPTER TWO: LITERATURE REVIEW

In this chapter, we will first introduce Deep Learning and CNN, how they work, their applications, and some SOTA architectures and frameworks we experimented with alongside datasets we use to train the model. Then, we will introduce the most advanced DNN compression strategies for edge computing that maximize performance while maintaining accuracy. Finally, we will discuss some methodologies for CNN-based hardware acceleration with HW-SW co-design approaches focusing on AMD ZYNQ SoC-FPGA with the PYNQ framework.

Fundamentals of Deep Learning

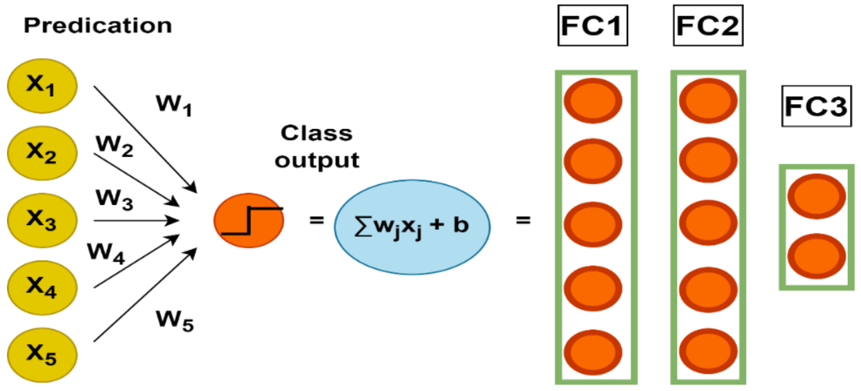

Figure 7. Deep Neural Network input and output mathematical equation

DNNs are a class of machine learning models inspired by the structure and function of the human brain. They consist of multiple layers of interconnected nodes, called neurons, which process and transmit information through the network [67]. The basic building blocks of DNNs are dense layers, also known as fully connected (FC) layers, which form the core of the network's architecture. In a dense layer, each neuron is connected to every neuron in the previous layer, forming a fully connected structure. The output of a neuron is mathematically computed by applying an activation function to the weighted sum of its inputs as shown in Fig. 7 and can be expressed as follows:

$$y = (f \sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

where y is the neuron output, f is the activation function, Wi is the weight associated with each input Xi, n is the number of inputs, and b is the bias term. The weights and biases are the learnable parameters of the network, which are adjusted during the training process using optimization algorithms such as gradient descent [68]. The weights determine the strength of the connections between neurons, while the biases allow for shifting the activation function to fit the data better. For example, consider a dense layer with three input neurons and two output neurons. The weight matrix W and bias vector b for this layer can be represented as follows:

$$W = \begin{pmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{2.2}$$

Given an input vector $x = [x_1, x_2, x_3]$, the output of the dense layer can be computed as follows:

$$y = f(Wx + b) = f\left(\frac{w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1}{w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2}\right) \tag{2.3}$$

The choice of activation function f depends on the specific requirements of the task and the desired properties of the network. Common activation functions shown in Fig. 8 include the Sigmoid Function, Hyperbolic Tangent (TANH), and Rectified Linear Unit (ReLU) [69]. By stacking multiple dense layers, DNNs can learn hierarchical representations of the input data, enabling them to capture complex patterns and relationships.

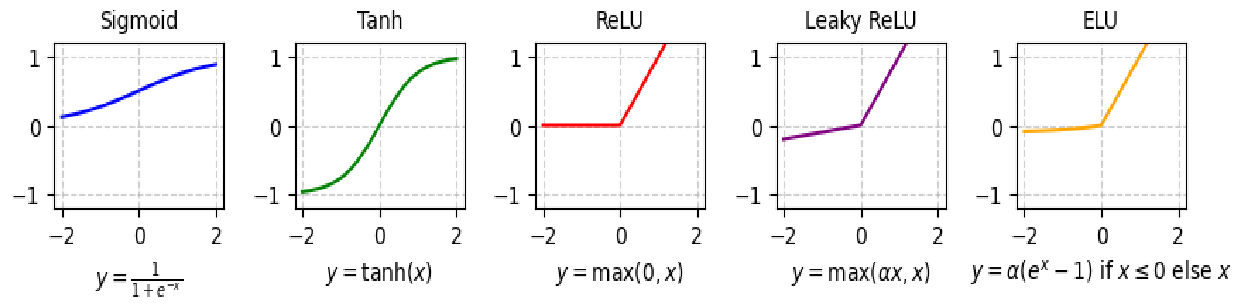

Figure 8. Different kind of activation function used while training DNN models.

Additionally, the backpropagation algorithm is used within network training, where predictions are compared to the ground truth labels using a loss function, such as the Mean Squared Error or Cross-Entropy Loss [70]. The goal is to minimize the loss function by adjusting the network's weights and biases. This is achieved by computing the gradients of the loss with respect to the parameters using the chain rule of derivatives. Let L be the loss function, and ∂y be the ground truth labels. Then, we compute the gradient of the loss with respect to the output activations as in the following equation follows:

$$\frac{\partial L}{\partial_a^{(L)}} = \frac{\partial L}{\partial y} \tag{2.4}$$

The depth of the network and the number of layers play a crucial role in its ability to learn intricate features and abstractions [71]. Besides, the feed-forward and backpropagation algorithms are repeated iteratively over multiple epochs until the network converges to a satisfactory solution [72].

Common DNN Framework and Datasets

The success of DNNs can be attributed to the development of powerful deep learning frameworks, such as TensorFlow [73] and PyTorch [74], which provide a wide range of tools and techniques for training and deploying DNN models. For example, Transfer Learning has become a crucial technique in training DNN models, especially when dealing with limited labeled data [75]. TensorFlow and PyTorch provide a wide range of pre-trained models that can be fine-tuned for specific vision tasks. Data augmentation is another essential technique for improving the model's generalization and robustness by applying random transformations to the training data [76].

On the other hand, the performance of DNN models is typically benchmarked on standard datasets, such as ImageNet [57] and MS COCO [77]. For instance, the current state-of-the-art model on the ImageNet dataset is OmniVec, achieving a top-1 accuracy of 92.4% [78]. Moreover, deploying DNNs on edge devices with limited computational resources and memory can be challenging. To address this issue, several datasets have been developed that are well-suited for benchmarking and evaluating the performance of DNNs on edge hardware, such as MNIST, CIFAR-10 shown in Fig. 9, and CIFAR-100 [79], Pascal VOC [80] shown in Fig. 10, and Tiny ImageNet [81]. These datasets have characteristics, such as small image sizes and a limited number of classes, that make them suitable for resource-constrained platforms. By using these datasets, researchers and practitioners can assess the accuracy and efficiency of compact DNN architectures and make informed decisions when deploying DNNs on edge hardware.

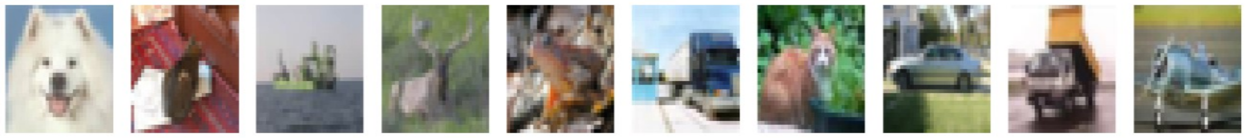

Figure 9. Subset of Cifar-10 dataset

Figure 10. Subset of Pascal VOC dataset.

Convolutional Neural Networks

CNNs as shown in Fig. 11 are a class of deep learning designed to process grid-like data, such as images and time series, by learning hierarchical representations through a series of convolutional, pooling, and FC layers [67]. The critical advantage of CNNs lies in their ability to exploit spatial locality and translation invariance, making them well-suited for vision tasks such as image classification, object detection, and semantic segmentation [71].

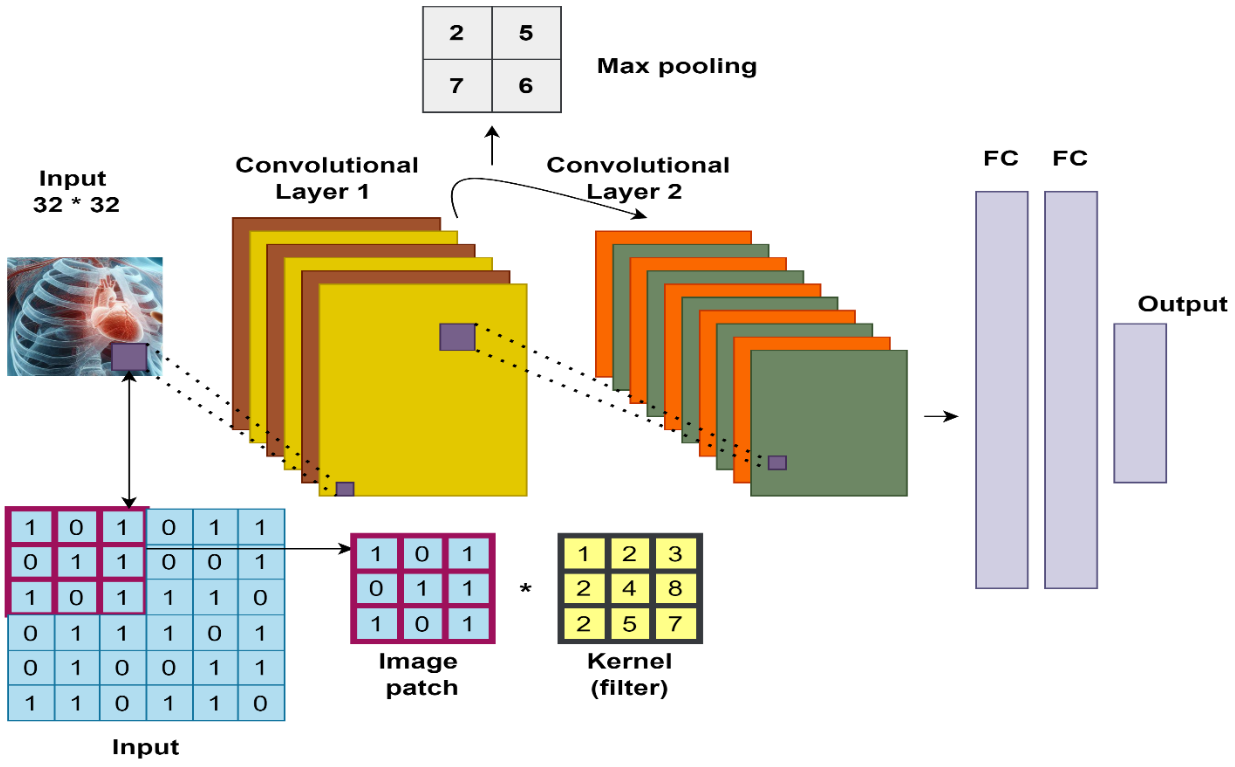

Figure 11. CNNs description including features extracting and pooling layers.

They are commonly represented in the RGB color space, where each pixel is described by three values corresponding to the intensities of red, green, and blue. However, the core of CNNs is convolutional layers, which consist of a set of learnable filters that slide across the input, performing element-wise multiplication using 3D convolutional kernels and summing the results to produce feature maps. The output of a convolutional layer can be mathematically expressed as follows:

$$y_i, j, k = f\left(\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{c=0}^{C-1} w_{m,n,c} * x_{i+m,j+n,c} + b_k\right) \tag{2.5}$$

Where $y_{i,j,k}$ is the output value at position (i, j) in the k-th feature map, f is a non-linear activation function (e.g., ReLU), $w_{m,n,c,k}$ is the weight at position (m, n) in the c-th input channel and k-th output channel, b_k is the bias term for the k-th output channel. $M, N, and\ C$ are the convolutional kernel's height, width, and depth, respectively. Moreover, the spatial dimensions of the output feature map can be calculated using the following equations:

$$H_{out} = \left[\frac{H_{in}+2P- K_h}{S_h}\right] + 1 \qquad (2.6) \qquad , \qquad W_{out} = \left[\frac{W_{in}+2P- K_w}{S_w}\right] + 1 \qquad (2.7)$$

Where H_{out} and W_{out} are the height and width of the output feature map, H_{in} and W_{in} are the height and width of the input feature map, P is the padding size, K_h and K_w are the height and width of the convolutional kernel. S_h and S_w are the stride values in the vertical and horizontal directions, respectively. Additionally, normalization comes after, which is a preprocessing technique used to standardize the input data before feeding it into the CNN [82]. Min-Max normalization is the most common normalization method that scales the pixel values to a fixed range from 0 to 1 using the following formula:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}} \quad (2.8)$$

Where x is the original pixel value, x_{min} and x_{max} are the minimum and maximum pixel values, respectively.

Pooling layers, on the other hand, are another essential component of CNNs. It down-sample the spatial dimensions of the feature maps while retaining the most relevant information [83]. The most common types of pooling operations are max pooling and average pooling, in which the max pooling can be expressed mathematically as follows:

$$y_{i,j,k} = \max_{m=0,n=0} M - 1, N - 1 \; x_i . s_{h+m,j} . s_{w+n,k} \quad (2.9)$$

Where $y_{i,j,k}$ is the output value at position (i, j) in the k-th feature map, $x_i . s_{h+m,j} . s_{w+n,k}$ is the input in the k-th channel, and M and N are the height and width of the pooling window, respectively. Moreover, fully connected layers are typically used at the end of the CNN architecture to perform high-level reasoning and classification [57]. These layers take the flattened output of the last convolutional or pooling layer and apply a linear transformation followed by a non-linear activation function. These layers perform classification, represented as:

$$y = f(W . x + b) \quad (2.10)$$

Where y is the output vector, f is a non-linear activation function (e.g., SoftMax for multi-class classification [84]), x is the input vector, W is the weight matrix, and b is the bias vector. Lastly, those combination of convolutional, pooling, and fully connected layers, along with carefully designing architectures and optimization techniques, enables CNNs to achieve remarkable performance on a wide range of computer vision tasks.

Computation Complexity of CNNs

CNNs have become the dominant architecture for various computer vision tasks due to their exceptional performance. However, the computational complexity of CNNs is a critical factor in their deployment, especially on resource-constrained devices. It is primarily determined by the number of floating-point operations (FLOPs) performed during a single forward pass. The total FLOPs in a CNN are the sum of the FLOPs in all convolutional, pooling, backbone, and other kind of layers. layers. The number of FLOPs in a convolutional layer can be calculated using the following equation:

$$FLOPs_{conv} = 2 \cdot H_{out} \cdot W_{out} \cdot C_{in} \cdot K_h \cdot K_w \cdot C_{out} \tag{2.11}$$

Where H_{out} and W_{out} are the height and width of the output feature map, C_{in} and C_{out} are the number of input and output channels, and K_h and K_w are the height and width of the convolutional kernel. For example, a layer transforming $224 \times 224 \times 3$ input into a $112 \times 112 \times 64$ output with a 3×3 kernel, the FLOPs calculation is:

$$FLOPs_{conv} = 2 \times 112 \times 112 \times 3 \times 3 \times 3 \times 64 = 86{,}704{,}128 \tag{2.12}$$

Fully connected layers also contribute to the computational complexity. The number of FLOPs is determined by the number of input and output neurons, as shown in the following equation:

$$FLOPs_{fc} = 2 \cdot N_{in} \cdot N_{out} \tag{2.13}$$

Where N_{in} and N_{out} are the numbers of input and output neurons. Moreover, the number of multiply-accumulate (MAC) operations is another metric used to measure the computational complexity of CNNs. MACs represent the number of multiplication and addition operations performed in a layer. For convolutional layers, the number of MACs is equal to the number of FLOPs divided by two, as shown in the equation follows:

$$MACs_{conv} = \frac{FLOPs_{conv}}{2} = H_{out} \cdot W_{out} \cdot C_{in} \cdot K_h \cdot K_w \cdot C_{out} \tag{2.14}$$

In addition to FLOPs and MACs, the number of parameters in a CNN also contributes to its computational complexity and memory requirements. It can be calculated as follows:

$$Parameters_{conv} = (K_h \cdot K_w \cdot C_{in} + 1) \cdot C_{out} \tag{2.15}$$

To put these metrics into perspective, let us consider the computational complexity of some popular CNN architectures. The VGG-16 model [58] has 138 million parameters and requires 15.5 billion FLOPs for a single forward pass. On the other hand, the MobileNet-V2 [85] architecture has only 3.5 million parameters and requires 300 million MACs, making it more suitable for resource-constrained devices. However, the high memory footprint makes deploying these networks on devices with limited memory resources challengng. Thus, reducing the computational complexity of CNNs is crucial for their deployment on edge devices and real-time applications.

Benchmarking the computational requirements and performance of DNNs and CNNs is crucial for assessing their suitability for applications and devices. The MLPerf benchmark [86] is a widely adopted benchmark suite that measures the performance of machine learning models across various tasks, including image classification, object detection, and semantic segmentation.

The benchmark provides a standardized way to compare the performance and efficiency of different hardware platforms and software frameworks. For example, the MLPerf benchmark results [87] as in Fig. 12 show that the NVIDIA A100 GPU can process 16,819 images per second on the ResNet-50 model for image classification. In comparison, the Google TPU v4 can process 27,366 images per second on the same model. These benchmarks help researchers and practitioners decide to select hardware and software platforms for their specific use cases. Lastly, various techniques have been proposed for edge computing and deployment to achieve this goal, such as pruning, quantization, and knowledge distillation where they will be discussed in the upcoming subsection.

Figure 12. MLPerf benchmark various DNNs among GPU, CPU, TPU, and FPGA.

State of the Art CNNs Architectures

When considering CNN architectures for edge computing, it is crucial to consider the limited computational resources, memory constraints, and power consumption. Edge computing refers to processing data locally smartphones, Internet of Things (IoT) devices, and embedded systems rather than sending the data to the cloud for processing [88]. This subsection will discuss CNN architectures for different vision tasks that are well-suited for edge computing, along with examples and metric benchmarking.

Image Classification: The EfficientNet architecture [7] shown in Fig. 13 (a) introduces a compound scaling method that uniformly scales the width, depth, and resolution. This allows for creating a family of models, from EfficientNet-B0 to EfficientNet-B7, with increasing complexity and accuracy. The EfficientNet-B7 model achieves a top-1 accuracy of 84.3% and a top-5 accuracy of 97.0% on ImageNet, surpassing the performance and efficiency of other state-of-the-art models. Moreover, the smaller versions of EfficientNet, such as EfficientNet-B0 and EfficientNet-B1, are particularly well-suited for edge computing due to their low computational complexity and memory requirements. It achieved a top-1 accuracy of 77.1% on the ImageNet dataset, with only 5.3 million parameters and 390 million FLOPs. This makes it an attractive choice for edge devices with limited computational resources and memory.

Object Detection: The YOLO (You Only Look Once) architecture [89] shown in Fig. 13 (b) has undergone several improvements. The latest version YOLOv8 [53] incorporates a modified version of the CSPDarknet53 [90] architecture forms the backbone and uses a Rectified Adam (RAdam) for improved optimization with Mix-up learning rate scheduler to achieve faster convergence and higher accuracy. One of the critical features of YOLOv8 is the use of a self-attention mechanism in the head that allows it to achieve an Average Precision (AP) of 53.9% on the MS COCO dataset at a real-time inference speed of 283 frames per second (FPS) on a Tesla A100 GPU. YOLOv8n is the most miniature version of YOLOv8 with only 3.2 million parameters and 4.6 billion FLOPs. It achieved an impressive mAP (mean Average Precision) of 37.3% on the COCO dataset.

Instance Segmentation: YOLOv8 has been extended by adding a specialized segmentation heads [91] to its architecture in parallel with the existing branch for bounding box recognition. These heads work alongside the detection components, leveraging multi-scale feature maps produced by the backbone and enhanced through feature fusion techniques like Feature Pyramid Networks (FPNs) or Path Aggregation Networks (PANs) [92]. This integration allows YOLOv8 to generate precise pixel-wise masks, making it adept at instance segmentation. Its backbone achieved an AP of 43.1% on the MS COCO dataset, for instance, segmentation while running at 248 FPS on a Tesla A100 GPU.

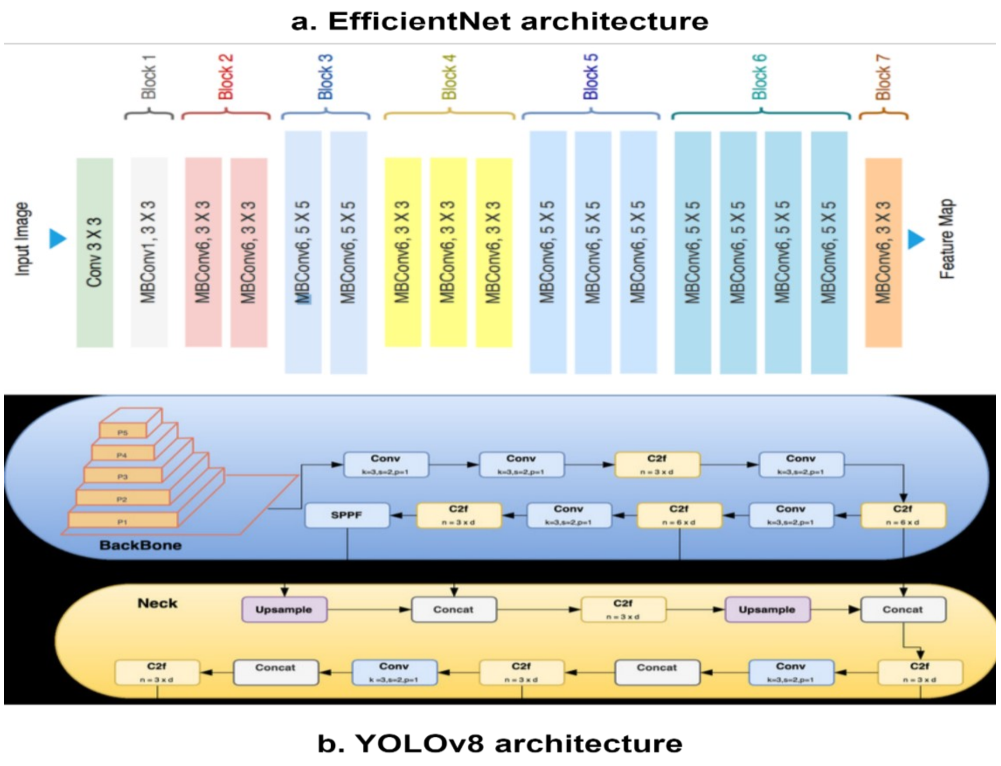

Figure 13. EfficientNet [7] and YOLOv8 architectures [53].

However, the performance of these architectures can be further improved by transfer learning, fine-tuning, and domain adaptation [93]. Transfer learning involves pre-training the models on large-scale datasets like ImageNet and fine-tuning them on specific tasks or domains. This approach can significantly reduce the training time and improve the performance, especially when dealing with limited labeled data. Both EfficientNet and YOLOv8 are two CNN architectures well-suited for edge computing due to their excellent balance between accuracy and efficiency, making them attractive choices for deployment on resource-constrained devices.

Deployment Challenges of CNNs on the Edge

Edge computing brings data processing closer to the source, offering benefits such as reduced latency and improved privacy [88]. However, deploying DNNs and CNNs on embedded systems with limited hardware resources, including data processing, memory bandwidth, and energy consumption, presents several challenges in some popular edge hardware platforms, such as Jetson, FPGA, ASIC, and Raspberry Pi. Those devices often have low-power CPUs, small amounts of RAM, and limited storage capacity, making running complex CNN models in real-time difficult. The depth and width of CNN networks increase the computational complexity, making it difficult to execute them efficiently. The choice of hardware platform depends on the specific application requirements, deployment scale, and budget constraints. To illustrate more, the number of FLOPs indicates the computation required to perform a single inference pass through the network. At the same time, memory bandwidth refers to the amount of data that can be transferred between the processor and memory per of unit time. To address these challenges, researchers have proposed various techniques for optimizing DNNs and CNNs for edge deployments, such as model Pruning, Quantization, Knowledge Distillation, and Neural

Architecture Search [94]. These techniques aim to reduce the models' computational complexity and memory footprint of the models while maintaining high accuracy.

Compression and Optimization of CNNs

Due to the significance important of efficient DNN for edge computing, extensive research has been conducted to enhance their accuracy, latency, and performance. This section will provide an organized overview of the prior studies. Initially, we examined previous endeavors in network compression, followed by enhanced precision achieved by regularization, and ultimately, we aim to expedite inference by leveraging hardware acceleration.

Pruning

Pruning as shown in Fig. 14 has emerged as a popular technique for compressing DNNs and CNNs by removing fewer essential weights, filters, or channels from the model, thereby reducing the model size and computational complexity while maintaining acceptable accuracy. Magnitude-based pruning is one of the most widely used techniques, which removes weights with small absolute values. This method is mathematically represented as follows:

$$W_P = \{w \in W : \mid w \mid > \theta\} \tag{2.16}$$

Where W is the set of all weights, θ is the threshold, and W_P is the set of weights after pruning. The efficiency of magnitude-based pruning is quantified using a Compression Ratio (CR), calculated as the ratio of the total number of weights to the number of pruned weights and expressed as follows:

$$CR = \left|\frac{W}{W_P}\right| \tag{2.17}$$

Han et al. [95] proposed a three-step pruning pipeline: training a dense model, pruning redundant connections based on weight magnitude, and retraining the pruned model to fine-tune the remaining weights. This method achieved a compression ratio of up to 9x for AlexNet and 13x for VGG-16 on the ImageNet dataset without significant loss in accuracy. They reported a top-5 accuracy of 80.3% for AlexNet and 89.1% for VGG-16 after pruning, compared to the original accuracies of 80.2% and 90.0%, respectively. However, magnitude-based pruning often results in irregular sparsity patterns, which are difficult to accelerate on hardware due to the overhead of handling sparse matrix operations.

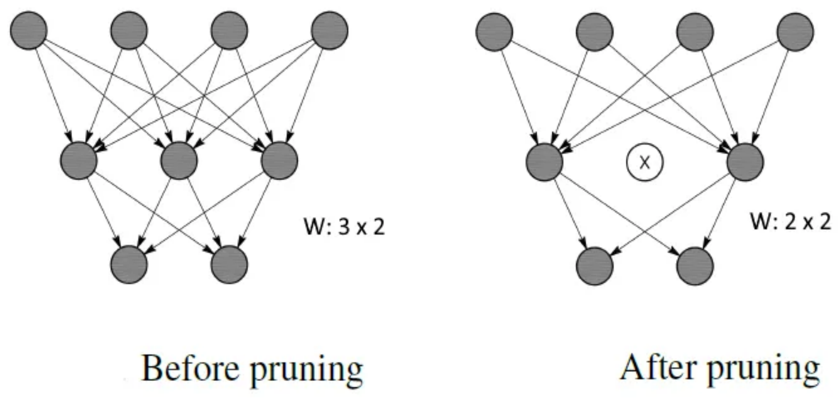

Figure 14. A well description of pruning processing before and after [96].

Structured pruning (filter pruning) is another method proposed to address the limitations of magnitude-based pruning by removing entire filters or channels from the CNN model. This method selects filters for removal whose L1-norm falls below a certain threshold α. The formula for identifying these filters is as follows:

$$F_P = \{F_i \in F : \| F_i \| 1 > \alpha\} \tag{2.18}$$

Where F represents the set of all filters in a layer, and F_P is the subset of filters retained after pruning. The compression ratio for filter pruning is then calculated by comparing the original number of filters to the pruned set as follows:

$$CR = \frac{n}{|F_P|} \tag{2.19}$$

Li et al. [97] proposed a filter pruning method that removes filters with small absolute weights and retrains the model to compensate for the accuracy loss. They achieved a 1.4× speedup for ResNet-110 on the CIFAR-10 dataset with only a 0.02% increase in error rate and reported an error rate of 6.45% for the pruned model, compared to 6.43% for the original model. Moreover, He et al. [98] proposed a channel pruning method that achieves a 2× speedup for VGG-16 on the ImageNet dataset with only a 0.3% decrease in top-5 accuracy. They reported a top-5 accuracy of 89.8% for the pruned model, compared to 90.1% for the original model.

Several advanced pruning techniques have also been proposed to improve the compression ratio and accelerate edge devices inference. Wen et al. [99] proposed a structured sparsity learning method that regularizes the structures of DNNs, such as filters, channels, and layers, to achieve high compression ratios and speedups. Liu et al. [100] proposed a network slimming method that learns channel-wise scaling factors and prunes channels with small scaling factors, achieving up to 20× compression on the VGG-16 model. Yang et al. [101] proposed a neural network accelerator architecture that leverages structured sparsity to achieve high performance and energy efficiency on FPGAs. Lastly, Lin et al. [102] proposed a framework for automatically pruning and deploying DNNs on resource-constrained edge devices, achieving up to 6.3× speedup on a Raspberry Pi.

Despite the remarkable progress made in CNNs pruning for edge computing, striking the right balance between model accuracy and computational efficiency remains a significant challenge [103]. To address these issues, future research should focus on developing more sophisticated pruning algorithms that can better understand and maintain the essential characteristics of the network [104]. This could involve leveraging advanced techniques from graph theory, information theory, and network science to analyze the connectivity patterns and information flow within the CNN, enabling more targeted and effective pruning strategies [105], [106]. Moreover, exploring dynamic pruning methods that can adapt to the edge device's current computational load in real-time offers another potential direction for future research [107].

Quantization

Quantization shown in Fig. 15 is a process that involves mapping a large set of continuous or high-precision values to a smaller set of discrete values. In the context of DNNs and CNNs, quantization is applied to weights and activations by converting the floating-point numbers (e.g., 32-bit floats) into lower-precision data types, such as integers (e.g., 8-bit integers) [6]. Uniform or post-training quantization is the simplest and most commonly used method for quantizing weights and activations. Given a real value x in the range $[a, b]$, the quantized value q can be obtained using the following equation:

$$q = round\left(\frac{x-a}{b-a} \cdot (2^n - 1)\right) \cdot \frac{b-a}{2^n-1} + a \tag{2.20}$$

where n is the number of bits used for quantization. The quantization error E can be calculated as follows:

$$E = x - q \tag{2.21}$$

Jacob et al. [108] proposed a uniform quantization scheme for quantizing weights and activations to 8-bit integers. They benchmarked their method on the ImageNet dataset using popular CNN architectures, such as MobileNet and Inception-V3. For MobileNet, they achieved a 4× compression ratio with only a 0.5% drop in top-1 accuracy compared to the full-precision model.

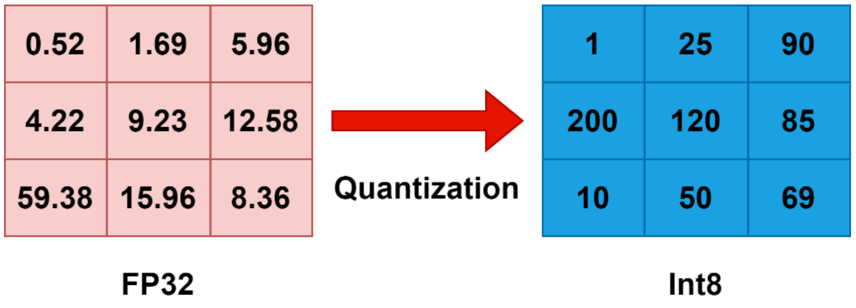

Figure 15. Quantization method used to convert a floating point into fixed point.

Moreover, quantization-aware training (QAT) is another method incorporated into the training process, allowing the model to adapt to reduced precision and mitigate accuracy loss. The forward pass uses quantized weights and activations during QAT, while the backward pass uses full-precision values. The gradients are computed concerning the quantized values using the straight-through estimator (STE) [109] as in the equation follows:

$$\frac{\partial L}{\partial W} \approx \frac{\partial L}{\partial W_q} \quad (2.22) \quad , \quad \frac{\partial L}{\partial A} \approx \frac{\partial L}{\partial A_q} \quad (2.23)$$

Where W and A represent the original weights and activations, respectively, while L denotes the loss function acting as a scaling factors. Krishnamoorthi [110] presented a comprehensive study of QAT techniques, including techniques for handling batch normalization and activation functions. They achieved a 4× compression ratio with less than 1% accuracy loss on the ImageNet dataset for ResNet-50 and MobileNet-V2.

Mixed-precision quantization is another technique that assigns different bit-widths to different layers or operations within the network based on their sensitivity to quantization errors [111]. This technique allocates varying bit-widths across different network layers or operations. For the $i-th$ layer with bit-width n_i, the quantized weight W_{qi} and activation A_{qi} are calculated as follows:

$$W_{qi} = round\left(\frac{W_i}{s_{wi}}\right) \cdot s_{wi} \quad (2.24), \qquad A_{qi} = round\left(\frac{A_i}{s_{ai}}\right) \cdot s_{ai} \quad (2.25)$$

Where s_{wi} and s_{ai} are the scaling factors for the $i-th$ layer, this approach allows for a more fine-grained trade-off between compression and accuracy. Wang et al. [112] proposed a hardware-aware automated quantization framework called HAQ, which jointly optimizes the bit-width and associated hardware accelerator design. They demonstrated a 4.1× speedup for MobileNet-V2 on an FPGA with less than 1% accuracy loss.

In summary, quantization is a technique that maps continuous or high-precision values to a smaller set of discrete values. This process is essential for reducing the model size and computational complexity of DNNs and CNNs, enabling their deployment on resource-constrained devices. Thus, using FX-Graph [113] Mode Quantization in PyTorch involves the implementation of quantization at the graph level, offering precise control over model compression. This method transforms a model into a graph of operations, enabling targeted quantization on specific subgraphs or nodes. It supports both automatic quantization and manual quantization, allowing for custom strategies. This makes it an essential tool for deploying deep learning models on edge devices while maintaining acceptable accuracy.

Knowledge Distillation

Knowledge distillation (KD) shown in Fig. 16 is a model compression technique that transfers knowledge from a large, complex teacher model to a smaller, more efficient student model, enabling the deployment of DNNs and CNNs on edge devices [114]. By leveraging the knowledge of the teacher model, the student network is trained to mimic the outputs of the more extensive network. These outputs (often class probability distributions) provide richer "soft targets" than complex compared ground-truth labels. The softmax temperature-based knowledge distillation is one type of soft target that involves training the student model to match the softened softmax outputs of the teacher model. The outputs are then obtained by dividing the logits (pre-softmax activations) by a temperature parameter T, which is given by:

$$P_T(x_i) = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j exp\left(\frac{z_i}{T}\right)} \tag{2.26}$$

Where $P_T(x_i)$ denotes the softened probability for class i, z_i is the logit corresponding to class i, and T is the temperature parameter. Moreover, the student model is trained to minimize the Kullback-Leibler (KL) divergence between its softened softmax outputs and those of the teacher model, as given in the following equation:

$$L_{KD} = aT^2 D_{KL}(P_T(x_{teacher})||P_T(x_{student})) + (1-a)L_{CE}(y_{true}, x_{student}) \tag{2.27}$$

Where L_{KD} is the overall knowledge distillation loss, D_{KL} is the KL divergence, L_{CE} is the cross-entropy loss, y_{true} are the actual labels, $x_{teacher}$ and $x_{student}$ are the logits of the teacher and student models, respectively, and α is a hyperparameter that balances the importance of the two loss terms.

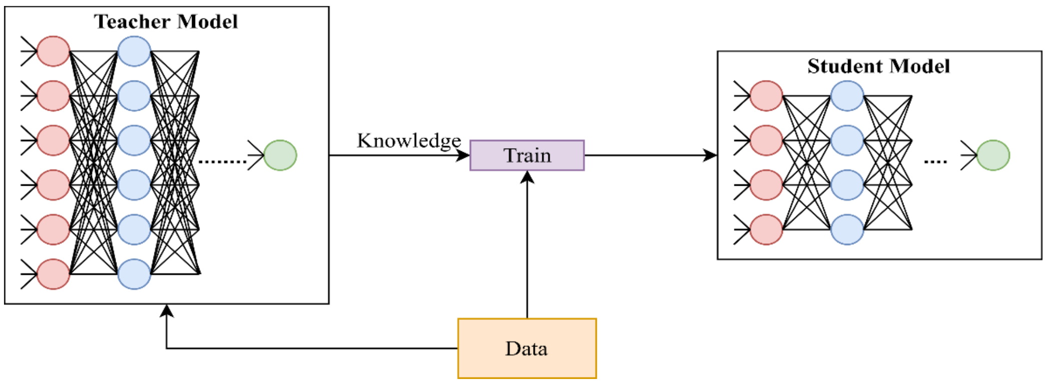

Figure 16. Knowledge Distillation method for high accuracy on the student model [115].

Hinton et al. [114] introduced this concept, demonstrated its effectiveness on the MNIST dataset. They achieved a 2.6% error rate using a student model with only 30% of the parameters of the teacher model. Additionally, Furlanello et al. [116] extended this approach and applied it to larger datasets, such as CIFAR-10 and CIFAR-100. They achieved a 2.5% error rate on CIFAR-10 using a student model with only 50% of the parameters of the teacher model.

In addition to matching the softmax outputs, Probabilistic KD methods aim to transfer the knowledge of the teacher model in the form of probability distributions. Malinin et al. [117] proposed a method called Uncertainty-Aware Knowledge Distillation (UAKD), which models the uncertainty of the teacher model's predictions using a Dirichlet distribution. The Dirichlet distribution is parameterized by a concentration parameter α, which is learned by the teacher model. Then, the student model is trained to match the concentration parameter of the teacher model as in equation follows:

$$L_{UAKD} = D_{KL}(Dir(a_{teacher})||Dir(a_{student})) \tag{2.28}$$

Where L_{UAKD} is the uncertainty-aware knowledge distillation loss, Dir denotes the Dirichlet distribution, and $a_{teacher}$ and $a_{student}$ are the concentration parameters of the teacher and student models, respectively. Malinin evaluated its performance on the CIFAR-10 and CIFAR-100 datasets. They achieved a 3.89% error rate on CIFAR-10 and an 18.22% error rate on CIFAR-100 using student models with only 50% of the parameters of the teacher models.

Quantized KD is a popular technique for reducing the memory footprint and computational complexity of DNNs and CNNs. Polino et al. [118] proposed a method called Quantized Distillation (QD), combining KD with quantization to achieve greater compression and efficiency. In QD, the teacher model is first quantized using a quantization function Q which can be expressed mathematically as follows:

$$W_{q_{teacher}} = Q(W_{teacher}) \tag{2.29}$$

Where $W_{teacher}$ and $W_{q_{teacher}}$ are the original and quantized weights of the teacher model, respectively. The student model is then trained to minimize the combined loss as follows:

$$L_{QD} = \beta L_{KD}(x_{teacher}, x_{student}) + (1-\beta) L_{CE}(y_{true}, x_{student}) \tag{2.30}$$

Where L_{QD} is the quantized distillation loss, L_{KD} is the knowledge distillation loss, L_{CE} is the cross-entropy loss, and β is a hyperparameter balancing the two loss terms. Polino evaluated its performance on the CIFAR-10 and ImageNet datasets. For CIFAR-10, they used a ResNet-18 teacher model and a student model with the same architecture. The teacher model achieved an accuracy of 94.8%, while the student model achieved an accuracy of 94.2% with 8-bit quantization and 93.7% with 4-bit quantization. This demonstrates that QD can maintain high accuracy even with aggressive quantization.

KD has proven to be an effective technique for compressing DNNs and CNNs. However, several challenges associated with KD compression methods that need to be addressed to improve their performance and applicability. First, architecture selection is a crucial challenge in KD. Choosing the optimal architecture for the student model is essential for achieving the desired compression rate while maintaining sufficient capacity to learn from the teacher model [119]. Next, KD methods often focus on transferring low-level features or class probabilities, which may not capture the high-level semantic information learned by the teacher model [120]. To solve those issues, automated architecture search is a promising direction for future improvements in KD. Developing automated methods for searching the optimal student model architecture based on the teacher model's characteristics and the target compression rate could streamline the KD process [121]. Adaptive hyperparameter tuning is another potential improvement, where adaptive mechanisms for automatically adjusting the hyperparameters during training could reduce the need for manual tuning and improve the robustness of KD methods [122]. In addition, distributed and parallel knowledge distillation is an essential area for future improvements, as it could enable their application to large-scale datasets and models, reducing the computational burden and memory requirements [123]. Finally, integrating KD with other compression techniques, such as pruning, quantization, and low-rank factorization, could lead to more effective and compact models suitable for deployment on edge devices [124].

Hardware-Software CNNs Inference Accelerator Design

Hardware-Software co-design with Deep Learning Accelerators (DLA) plays a crucial role for the rapid growth and widespread adoption of AI applications in computer vision and Large Language Model (LLM). These specialized hardware architectures are designed to execute the complex mathematical operations involved in DNNs efficiency, providing significant performance and energy efficiency compared to general-purpose processors such as CPU [125]. The increasing demand for real-time inference in computer vision applications brings computation and data storage closer to the source of data, thereby reducing latency and bandwidth requirements [126]. Several key metrics are commonly used when comparing DLAs. These include:

- Throughput: It is used for evaluating the raw performance of the accelerator and its ability to handle large-scale workloads. It can be measured in frames per second (fps).
- Latency: It refers to the time taken by the accelerator to process a single input sample or perform a specific operation.
- Energy Efficiency: It is a critical metric for edge devices DLA. It is measured in terms of operations per second per watt (OPS/W) or samples per second per watt (samples/s/W).
- Flexibility: It refers to the ability of the accelerator to support different DNN architectures, data types, and optimization techniques. A flexible accelerator can adapt to evolving DNN models and application requirements, making it more future-proof.
- Scalability: It measures the ability of the accelerator to maintain performance as the size and complexity of the DNN models increase. Scalable accelerators can handle larger models and datasets without significant degradation in performance.

To evaluate these metrics, DLA can be bench-marked using the MLPerf benchmark suite [127]. Another popular benchmark and standardization efforts is the MLCommons [128]. It aims to provide a collaborative platform for developing and maintaining benchmark suites, ensuring fair and consistent comparisons across different accelerators among different vendors.

On the other hand, one of the primary challenges in DLA is the vast computational and memory requirements of DNNs. To address this, many accelerators employ specialized processing units, such as systolic arrays or tensor cores, which are optimized for the matrix-matrix and matrix-vector operations prevalent in DNNs [129]. These processing units are often arranged in a spatial architecture, where the processing elements (PEs) are arranged in a two-dimensional array. Each PE typically consists of a MAC unit, a local memory, and a control unit. The spatial arrangement allows for parallel computation and efficient data reuse, reducing the need for frequent memory accesses [130]. For example, the TPU developed by Google features a 256x256 systolic array that can perform 65,536 MAC operations per clock cycle [27]. Similarly, the Eyeriss accelerator employs a spatial architecture with 168 PE arranged in a 12x14 grid, enabling efficient computation of convolutional layers [40]. In terms of performance characteristics, the TPU has shown a 15-30x speedup over CPUs and GPUs on inference tasks. The Eyeriss accelerator has also achieved a power efficiency of 278 GOPS/W on convolutional layers, which is 2-3 orders of magnitude higher than CPUs and GPUs. However, the performance of DLA can vary depending on the specific DNN model, dataset, and hardware configuration. On the other hand, DLA typically incorporate high-bandwidth memory systems, such as high-bandwidth memory (HBM) or on-chip scratchpad memory, to minimize data movement and improve overall performance [131]. Besides, DLA often incorporate flexible dataflow mechanisms, allowing for scheduling of data movement and computation based on the specific characteristics of the target DNN [132].

DLA can be implemented by leveraging different hardware acceleration platforms such as GPUs, FPGAs, and ASICs to optimize the execution of neural network workloads [32]. These platforms offer massive parallelism, high memory bandwidth, and energy efficiency, making them suitable for the compute-intensive and data-parallel nature of CNNs. However, each platform has its own advantages and limitations in terms of performance, flexibility, and development complexity [6]. GPU with NVIDIA's CUDA programming model and cuDNN library have made it easier to develop and optimize CNN applications on GPUs [133]. However, GPUs have limitations in terms of power consumption and may not be suitable for low-power embedded systems. ASICs offer high performance and energy efficiency but the high non-recurring engineering costs and longer design cycles associated with ASICs make them suitable for high-volume production and specific application domains [27]. Examples of ASIC-based CNN accelerators include Google's TPU and Intel's Neural Network Processor (NNP) [134]. However, FPGAs have become a popular choice for DLA at the edge due to their reconfigurability, flexibility, and rapid prototyping capabilities [135]. FPGA-based accelerators can be customized to match the specific requirements of a given deep learning model, enabling efficient resource utilization and reduced power consumption [136]. Moreover, HLS tools, such as AMD Vitis-HLS and Intel OpenCL SDK for FPGAs, allow developers to describe hardware functionality using high-level programming languages, facilitating rapid prototyping and design space exploration [137]. On the other hand, SoC-based DLA have also gained traction in edge computing scenarios, as they offer a balance between performance, energy efficiency, and flexibility. These accelerators integrate multiple processing units, such as CPUs, GPUs, and dedicated neural processing units (NPUs), on a single chip, enabling efficient data sharing and communication. The heterogeneous nature of SoC-based accelerators allows for the optimization of different tasks, such as data preprocessing, model inference, and post-processing, leading to improved overall system performance. By performing inference and decision-making close to the data sources, edge intelligence can reduce latency, improve privacy, and enable many applications in areas such as robotics and autonomous systems.

Nowadays, AMD has redefined the embedded systems landscape by merging a high-performance CPU with a flexible FPGA fabric to create the heterogeneous ZYNQ SoC-FPGA architecture shown in Fig. 17. This innovative architecture combines the flexibility of programmable logic with the performance of a high-speed processing system, making it an excellent choice for deep learning accelerators. This architecture consists of two main components: the PS and the PL [138]. The PS is built around a dual-core ARM Cortex-A9 processor, which provides a familiar and powerful computing environment for running operating systems, software frameworks, and applications. The PL, on the other hand, is a flexible FPGA fabric that can be customized and programmed to implement hardware accelerators and custom interfaces using HLS [139]. it provides a rich set of on-chip block Random Access Memory (BRAMs) resources, Look-Up Table (LUT), Digital Signal Processing (DSP) slices, and high-speed interconnects bus; named AXI alongside DMA, which are essential for implementing efficient DLA. Another advantage of the ZYNQ SoC-FPGA for DLA is its energy efficiency. FPGAs are known for their low power consumption compared to other accelerator platforms, such as GPUs. This is particularly important for edge computing scenarios, where energy efficiency is a critical consideration.

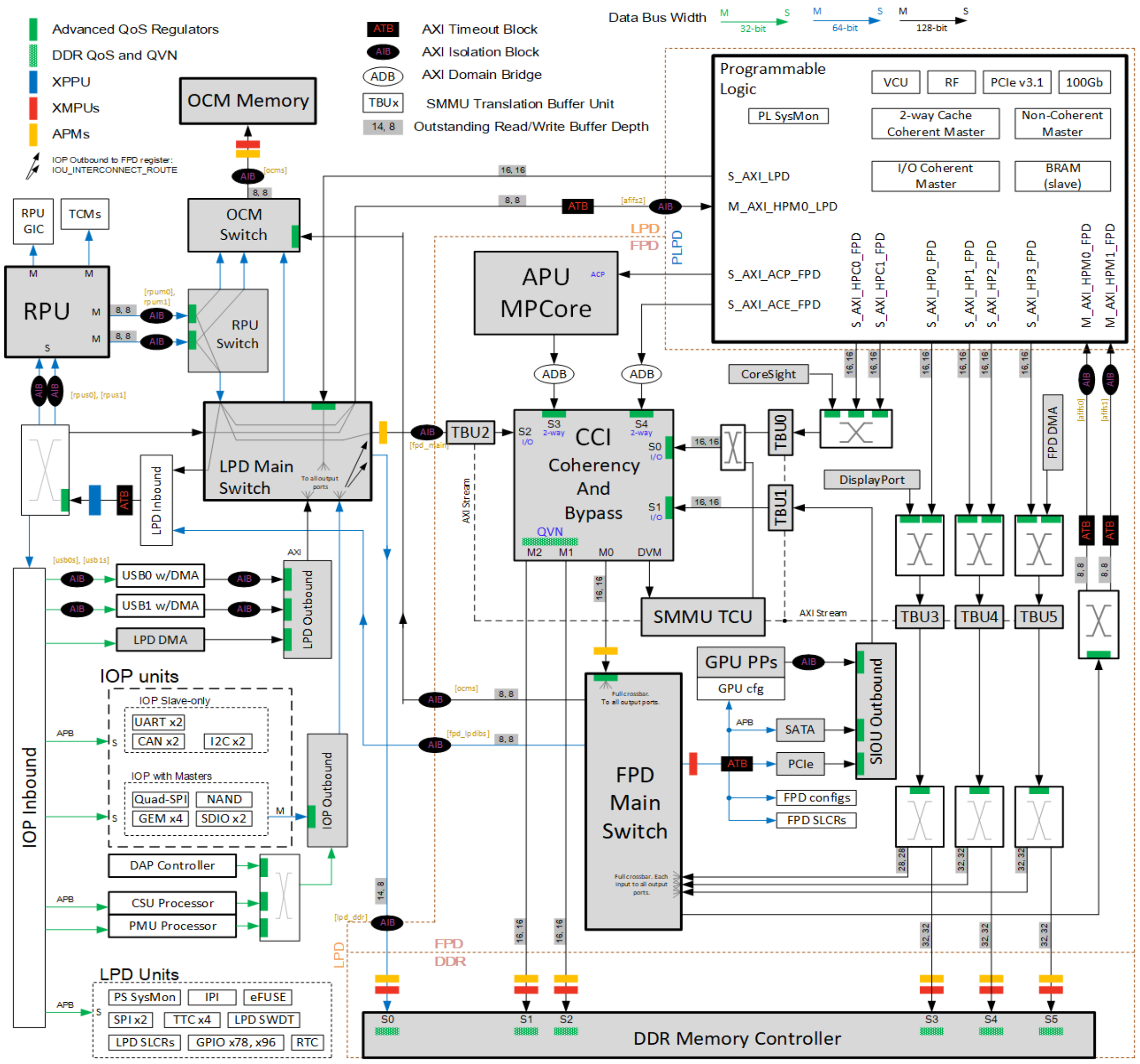

Figure 17. AMD ZYNQ Ultrascale+ [140] architecture internal details used to implement CNNs.

Since CNNs are computationally intensive and require significant processing power, the ZYNQ CPU part can handle the overall control flow, data management, and high-level operations, while the FPGA fabric can be utilized to implement custom hardware accelerators and exploit inherent parallelism for CNNs that involve a large number of matrix multiplications through CLB as shown in Fig. 18. This is done through:

- Utilizing DSP slices, which are optimized for mathematical operations commonly found in DNNs algorithms such as MAC operations.
- Enabling DMA with AXI interface for fast data transfer between the PS and the PL, as well as between different hardware modules within the FPGA fabric.
- BRAMs can be used as on-chip buffers to store intermediate results and minimize external memory accesses, reducing latency and improving energy efficiency.
- LUTs can be configured to stores the truth table of the desired logical function in the Static RAM (SRAM) while the multiplexers can select the appropriate output based on the input combination.

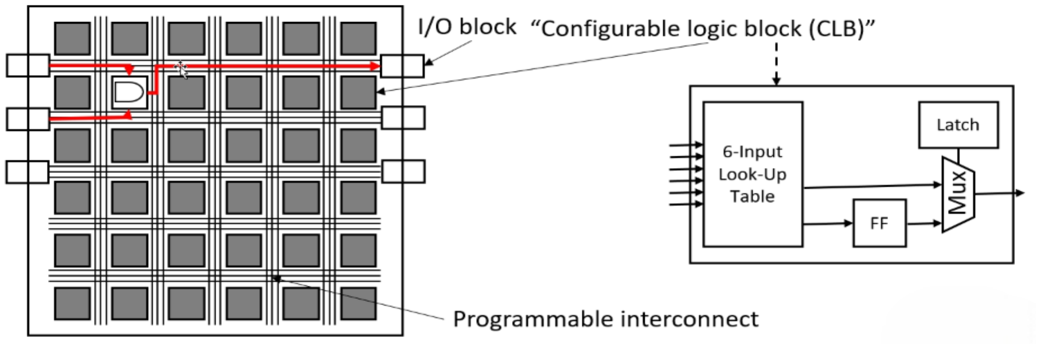

Figure 18. This is the internal structure of FPGA chip consists of building blocks (CLB) that do MAC operations.

However, there are also challenges associated when using ZYNQ SoC-FPGAs for DLA. One challenge is the development complexity involved and limited memory bandwidth in designing hardware accelerators on FPGAs [32]. While AMD provided HLS tools and frameworks that made the development process more accessible, there is still a learning curve associated with FPGA programming and optimization techniques. Additionally, the limited resources on FPGAs, such as memory and logic elements, can pose constraints on the size and complexity to store weights, activations, and intermediate results of the deep learning models that can be accelerated. Despite these challenges, the ZYNQ SoC-FPGA remains an attractive platform for edge DLA due to its unique combination of flexibility, performance, and energy efficiency. As the field of deep learning continues to evolve, with the emergence of new algorithms and architectures, the adaptability of ZYNQ SoC-FPGAs allows developers to quickly prototype and deploy high-performance DLA that are optimized for specific workloads [141].

Understanding CNNs Internal Structure for Edge Acceleration

Optimizing the CNNs architecture is crucial for maximizing throughput and efficiency when it comes to hardware acceleration. CNNs are composed of multiple layers, including convolutional layers, pooling layers, and fully connected layers, each performing specific operations on the input data [1]. The convolutional layers are the most computationally intensive, as they perform a large number of MAC operations alongside floating point to convolve the input feature maps with learnable filters [2]. To maximize CNN throughput, it is essential to optimize the MAC computation and compress the model, which is the primary bottleneck in CNN inference [3]. it is also essential to consider the trade-offs between accuracy and computational complexity. By carefully selecting the number and size of filters, the depth of the network, and the type of activation functions, it is possible to strike a balance between model performance and inference speed.

Analysis CNN For Multiply and Accumulate Operations

MAC operations are the core computational units in Convolutional Neural Networks (CNNs). In a CNN, the majority of the computations occur in the convolutional layers, where the input feature maps are convolved with a set of learnable filters to produce output feature maps. Each convolutional operation involves a large number of MAC operations, which can be a significant bottleneck in terms of computational complexity and energy consumption, especially on resource-constrained edge devices. The number of MAC operations required for a single convolutional layer can be calculated using the following formula [40]:

$$MAC_{ops} = \frac{K_h \times K_w \times C_{in} \times C_{out} \times H_{out} \times W_{out}}{stride^2} \tag{2.31}$$

Where K_h and K_w are the height and width of the convolutional kernel respectively, C_{in} and C_{out} are the number of input and output channels, H_{out} and W_{out} are the height and width of the output feature map, and $stride$ is the stride of the convolution. This formula act as an evident that the number of MAC operations scales linearly with the number of input and output channels, as well as the spatial dimensions of the output feature map. Therefore, reducing the number of channels and the spatial dimensions can significantly reduce the computational complexity of CNNs.

To optimize more the internal structure of CNNs for MAC computation shown in Fig. 19, several techniques have been proposed in recent years. These techniques aim to reduce the computational complexity and memory footprint of CNNs while maintaining their accuracy and performance. One popular approach is to use depthwise separable convolutions, which decompose the standard convolution operation into a depthwise convolution followed by a pointwise convolution. This technique significantly reduces the number of parameters and MAC operations required, making it suitable for resource-constrained edge devices. MobileNet [142] and Xception [143] are examples of CNN architectures that employ depthwise separable convolutions to achieve high efficiency. Another technique for optimizing CNN internal structure is to use group convolutions, which divide the input channels into groups and perform convolutions within each group independently. This approach reduces the computational complexity and memory bandwidth requirements of CNNs, as demonstrated in architectures like ResNeXt [144] and ShuffleNet [145]. By carefully designing the grouping strategy and combining it with channel shuffling, these architectures achieve a good balance between accuracy and efficiency.

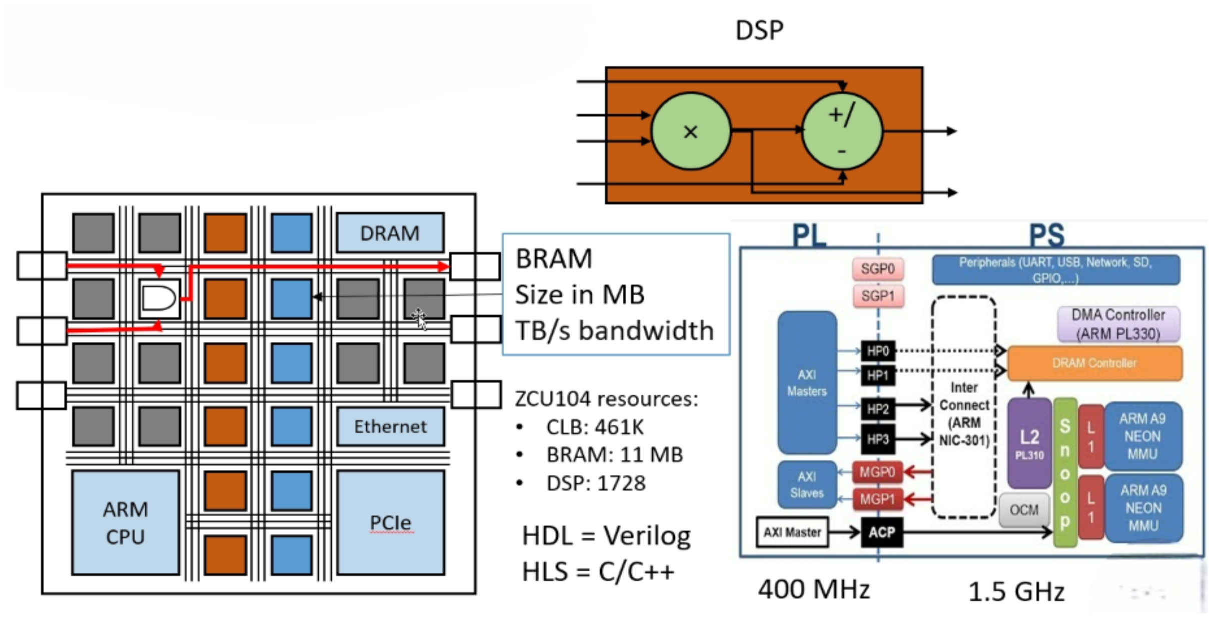

Figure 19. The ZYNQ architecture shown the internal parts that handle CNN computation.

Structured sparsity is another promising approach for optimizing CNN internal structure. By inducing sparsity in the convolutional filters, the number of non-zero weights and MAC operations can be significantly reduced [146]. Techniques such as channel pruning [147], filter pruning [148], and block-based sparsity [149] have been proposed to exploit structured sparsity in CNNs. These techniques not only reduce the computational complexity but also facilitate efficient hardware implementations by leveraging the regular structure of the sparse filters. Moreover, Low-bit quantization is also an effective technique for optimizing CNN internal structure. By quantizing the weights and activations of CNNs to lower bit widths, the memory footprint and MAC computation can be significantly reduced. Binarized Neural Networks (BNNs) [150] and Ternary Weight Networks (TWNs) [151] are extreme cases of quantization, where the weights and activations are represented using only 1 or 2 bits. These quantized networks can be efficiently implemented on hardware using bitwise operations and lookup tables, resulting in substantial computational and memory savings. In addition to these techniques, Batch normalization [152] is another technique that normalizes the activations of each layer, which not only improves the training stability but also allows for higher learning rates and faster convergence.

Lastly, by carefully designing and applying these techniques, it is possible to achieve significant reductions in computational complexity and memory footprint while maintaining the accuracy and performance of CNNs inference on edge devices.

Methodology of SoC-FPGA-based CNNs Inference Accelerators Design

The design methodology of CNNs inference accelerators involves various aspects of the system architecture as shown in Fig. 20, including the PS, PL, PEs, single instruction multiple data (SIMD) units, AXI, DMA, fixed-point arithmetic, double data rate (DDR) memory, and on-chip BRAM weight buffers. The design also requires careful consideration of frequency, energy efficiency, and educing memory access and memory bandwidth to achieve optimal throughput while minimizing power consumption [33].

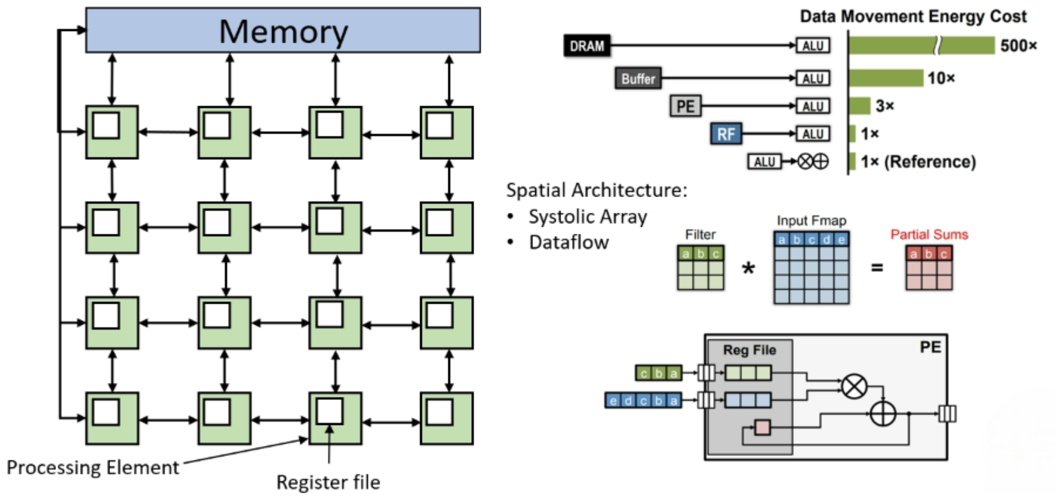

Figure 20. CNN-FPGA architecture design used to accelerate the inference.

The design methodology involves partitioning the CNN workload between the PS and PL, leveraging the strengths of each component. The PS is responsible for scheduling and coordinating the execution of the CNN layers, while the PL accelerates the convolutional, pooling, and fully connected layers [153]. The PS can be used to implement the softmax and classification layers, which are less computationally intensive than the convolutional and fully connected layers. On the other hand, the PL is ideal for accelerating the computationally intensive operations, such as convolutions and matrix multiplications, which can be parallelized and pipelined for high throughput [154]. The communication between the PS and PL is facilitated by the AXI interconnect, which provides high-bandwidth, low-latency data transfer [155]. The AXI interconnect supports various communication protocols, such as AXI4-Full, AXI4-Lite, and AXI4-Stream, which can be used for different types of data transfer, such as memory-mapped, register-based, and streaming interfaces, respectively. The design methodology involves selecting the appropriate AXI protocol and configuring the AXI interconnect to optimize the data transfer between the PS and PL, taking into account the bandwidth and latency requirements of the CNN workload.

The PL of the SoC-FPGA is used to implement an array of PEs that perform the core computations of the CNN [130]. Each PE typically consists of MAC units, which perform the dot product operations between the input activations and the weights [154]. The number and configuration of the PEs depend on the available resources of the FPGA, the performance requirements of the CNN workload, and the power and energy constraints of the system. Moreover, the PEs are designed to efficiently compute the innermost loop MAC operations. The number of MAC operations required for a single output feature map can be expressed as:

$$MAC_{ops} = N \times M \times P \times Q \times K \times R \times S \tag{2.32}$$

Where N is the batch size, M is the number of output channels, P and Q are the dimensions of the output feature map, K is the number of input channels, and R and S are the dimensions of the convolutional kernel. To exploit the parallelism in CNN computations, the PEs are often organized into SIMD units, which allow multiple MAC operations to be performed simultaneously. The number of MAC operations that can be performed in parallel by a SIMD unit of width W is as follows:

$$SIMD\_MAC_{ops} = W \tag{2.33}$$

And, the number of SIMD units required to compute a single output feature map is as follows:

$$SIMD_{units} = \frac{KxRxS}{W} \tag{2.34}$$

The SIMD units can be implemented using the DSP slices or the LUTs of the FPGA, depending on the precision and throughput requirements of the CNN workload. The PEs can be arranged in a single layer or multiple layers, depending on the depth and width of the CNN. In a single-layer arrangement, the PEs are connected in a pipeline, with each PE performing a single MAC operation per clock cycle. In a multi-layer arrangement, the PEs are connected in a mesh or a systolic array, with each PE performing multiple MAC operations per clock cycle. The total number of PE cycles required to compute a single output feature map is as follows:

$$PE_{cycles} = \frac{NxMxPxQ}{SIMD_{units}\, x\, PE_{utilization}} \tag{2.35}$$

Where $PE_{utilization}$ is the utilization factor of the PEs, dependent on dataflow and memory access patterns.

On the other hand, DMA can be used to minimize the overhead of data transfers which allows the PL to access the PS memory directly without the involvement of the processor. DMA enables fast and efficient data transfer between the PS and PL, reducing the latency and increasing the throughput of the CNN workload. The DMA engine can be implemented using the DMA controller IP provided by the FPGA vendor or using custom DMA engines optimized for the specific requirements of the CNN workload. The DMA engine can be configured to support various transfer modes, such as single transfer, burst transfer, and circular transfer, depending on the data access patterns of the CNN layers. The design methodology involves selecting the appropriate DMA configuration and optimizing the data transfer parameters, such as the burst size and the transfer width, to minimize the latency and maximize the throughput of the data transfers.

Additionally, CNN inference accelerators often use a fixed-point arithmetic format for the weights and activations to reduce the hardware complexity and improve the performance and energy efficiency compared to floating-point arithmetic [156]. Fixed-point arithmetic represents numbers with a fixed number of bits for the integer and fractional parts, which simplifies the hardware implementation and reduces the memory footprint. The choice of fixed-point format involves a trade-off between accuracy and efficiency. Using a higher number of bits for the integer and fractional parts can improve the accuracy of the CNN inference, but it also increases the hardware complexity and reduces the performance and energy efficiency, and vice versa [157]. Techniques such as quantization can be used to optimize the fixed-point format. Quantization involves mapping the floating-point weights and activations to a set of discrete values, which can be represented using a fixed number of bits. The quantization scheme can be uniform or non-uniform, depending on the distribution of the weights and activations.

SoC-FPGAs typically include external DDR memory, which serves as the main storage for the input data, output data, and intermediate feature maps of the CNN [44]. DDR memory provides high capacity and bandwidth, but accessing it incurs significant latency and energy overhead. The bandwidth (BW) of the DDR memory can be expressed as follows:

$$BW_{DDR} = f_{DDR} \; x \; W_{DRR} \; x \; N_{DDR} \tag{2.36}$$

Where f_{DDR} is the clock frequency of the DDR memory, W_{DRR} is the width of the DDR memory interface, and N_{DDR} is the number of DDR memory channels. Techniques such as tiling, caching, and prefetching can be used to optimize the memory hierarchy [153]. Tiling involves partitioning the input and output data into smaller tiles that can fit into the on-chip memory of the FPGA, reducing the need for frequent DDR memory accesses. The tiling parameters, such as the tile size and the tiling order, can be determined using techniques such as loop tiling and data tiling, which optimize the data locality and minimize the data movement. Caching involves storing frequently accessed data in on-chip buffers, such as BRAMs or distributed RAMs (DRAMs), which have lower latency and energy overhead compared to DDR memory. It can be determined using techniques such as working set analysis and cache modeling, which estimate the temporal and spatial locality of the data accesses. Prefetching involves fetching the data from DDR memory in advance, before it is needed by the computation, hiding the latency of the memory accesses. It can be determined using techniques such as data flow analysis and access pattern prediction, which estimate future data access patterns based on past accesses.

To reduce the latency and energy overhead of accessing the weights from the DDR memory, SoC-FPGA-based CNN inference accelerators often include on-chip weight buffers. These buffers are implemented using the fast on-chip memory resources of the FPGA, such as BRAMs [158]. The on-chip weight buffers store a subset of the weights that are frequently accessed by the PEs, reducing the need for off-chip memory accesses. The size of the on-chip weight buffer can be expressed as follows:

$$S_{buffer} = N_{PE}\ x\ S_{PE} \tag{2.37}$$

Where N_{PE} is the number of PEs and S_{PE} is the size of the weight buffer per PE. The size of the weight buffers can be determined using techniques such as working set analysis and buffer sizing, which estimate the minimum amount of memory needed to store the frequently accessed weights. The organization of the weight buffers can be determined using techniques such as data layout optimization and memory partitioning, which optimize the data placement and minimize memory conflicts. The weight buffers can be organized in various ways, depending on the dataflow and the computation order of the CNN layers. For example, the weights can be stored in a row-major or column-major order, depending on the access patterns of the convolutional and fully connected layers. The weights can also be stored in a compressed format, using techniques such as sparse matrix compression and run-length encoding, which reduce the memory footprint and the memory bandwidth requirements. The number of weight buffer accesses required for a single convolutional layer can be expressed as follows:

$$N_{access} = \frac{K\ x\ R\ x\ S}{S_{buffer}} \tag{2.38}$$

Additionally, the buffer organization can be determined using techniques such as data reuse analysis and access pattern optimization, which maximize the data locality and minimize the data movement [154]. The compression scheme can be determined using techniques such as sparsity analysis and compression ratio estimation, which estimate the compression efficiency and the decompression overhead. In addition to the on-chip weight buffers, SoC-FPGA-based CNN inference accelerators often include on-chip activation buffers and output buffers, which store the intermediate feature maps and the output feature maps, respectively. Nevertheless, throughput can represent the amount of data that the accelerator can process per unit of time, typically measured in operations per second (OPS) or frames per second (FPS). Higher throughput indicates faster execution of CNN workloads, which is essential for real-time applications such as video processing, autonomous driving, and robotics [66]. The throughput of a CNN accelerator can be calculated using the following equation:

$$Throughput = \frac{Batch_{size}\ x\ Clock_{frequency}}{Cycles\ per\ batch} \tag{2.39}$$

Where $Batch_{size}$ is the number of inputs processed in each batch, $Clock_{frequency}$ is the operating frequency of the accelerator in Hz, and $Cycles\ per\ batch$ is the number of clock cycles required to process each batch. However, to achieve high throughput, CNN accelerators must be designed with efficient hardware architectures and memory hierarchies that can exploit the parallelism and data reuse opportunities in CNN workloads. This includes techniques such as pipelining, data tiling, and caching, which can reduce the latency and memory bandwidth requirements of CNN operations.

As the field of CNN acceleration continues to evolve, new techniques and architectures are emerging to address the challenges of performance, efficiency, and scalability. These include the use of more advanced process technologies, such as 3D integration and non-volatile memory, as well as the exploration of new computing paradigms, such as in-memory computing and neuromorphic computing. Ultimately, the success of SoC-FPGA-based CNN inference accelerators will depend on the ability of designers to keep pace with the rapidly evolving landscape of CNN workloads and hardware platforms and to develop innovative solutions that can meet the growing demands for intelligent and efficient computing at the edge.

Strategies for Accelerating CNN Inference on FPGA

Optimizing SoC-FPGA-based CNN inference accelerators involves a systematic approach considering both hardware and software aspects. Hardware acceleration techniques aim to exploit the parallelism and data reuse opportunities in CNNs by utilizing spatial and temporal architectures to optimize the accelerator datapaths' performance, energy efficiency, and resource utilization.

These techniques include the use of systolic arrays [159], dataflow architectures [160], and memory hierarchy optimizations [154] to minimize data movement and maximize computational throughput. Additionally, algorithmic optimizations, such as loop unrolling, loop tiling, and data layout transformations [45], can be applied to improve the utilization of hardware resources and minimize data transfer overhead. HLS shown in Fig, 21 has also emerged as a powerful tool for accelerating the development of CNN accelerators on Zynq SoC-FPGAs [34]. HLS allows designers to describe the accelerator architecture and functionality using high-level programming languages, such as C/C++ and automatically generates the corresponding hardware description language (HDL) code [161]. HLS enables rapid prototyping, design space exploration, and optimization of CNN accelerators, reducing the development time and effort compared to traditional HDL-based approaches [38].

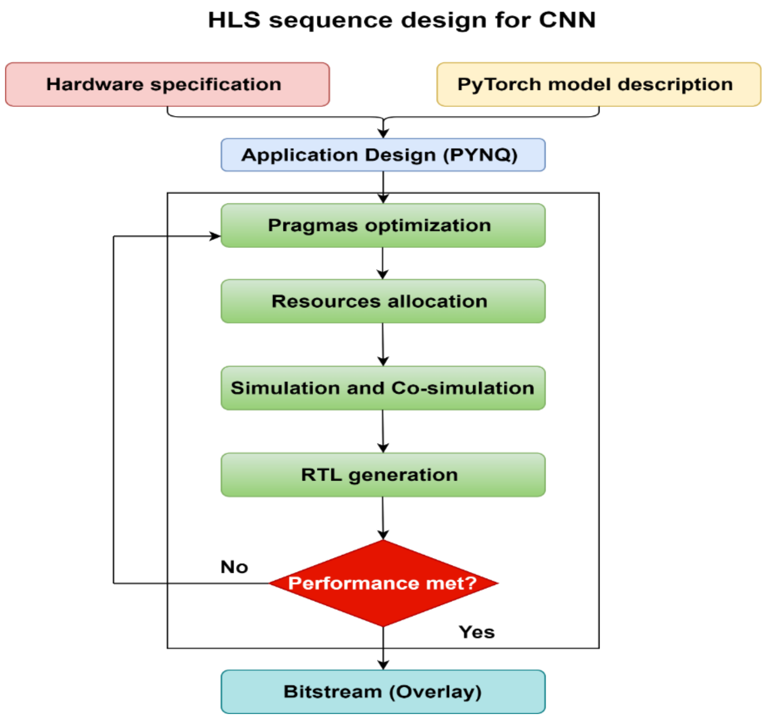

Figure 21. The High-Level Synthesis sequence design for CNN deployment on FPGA.

The combination of hardware acceleration techniques, software optimizations, and HLS-based design methodologies has significantly accelerated CNN inference on Zynq SoC-FPGAs. This subsection will explore various strategies for accelerating CNN inference and investigate the trade-offs between performance, energy efficiency, and resource utilization. Through a comprehensive analysis and experimental evaluation, we aim to contribute to the state-of-the-art CNN acceleration on Zynq SoC-FPGAs and pave the way for deploying of intelligent edge devices in real-world applications.

Compression and Layer Fusion

Compression and layer fusion are two important techniques for optimizing the CNN model for inference. These techniques reduce computational complexity, memory footprint, and energy consumption while maintaining accuracy and performance. Compression techniques, such as pruning, quantization, and KD, can significantly reduce the size and complexity by removing redundant or less important parameters and representations. Layer fusion, on the other hand, involves merging multiple layers of the CNN model into a single layer, which can reduce the number of memory accesses and data movements between the layers [142]. This is particularly important for FPGA-based accelerators, where the on-chip memory and bandwidth are limited, and the off-chip memory accesses are expensive regarding latency and energy. The intermediate feature maps can be kept on-chip and reused by fusing the layers for multiple operations, reducing the need for off-chip memory accesses [160]. For example, Alwani et al. [162] proposed a layer fusion technique that automatically fuses a CNN model's convolutional and fully connected layers based on their data dependencies and resource constraints. They demonstrated that their technique reduced the resource utilization and DSP usage by 95%, compared to a baseline implementation without layer fusion.

Designing efficient compression and layer fusion techniques for CNN inference accelerators is a challenging task that requires careful consideration of the trade-offs between accuracy, performance, and resource utilization. However, the integration of compression and layer fusion techniques into the hardware architecture of the accelerator requires specialized design tools and methodologies that can automate the process of model optimization and hardware generation. AMD Vitis-HLS can facilitate the design of compressed and fused CNN accelerators by providing high-level abstractions and optimizations for hardware implementation [137].

Loop Unrolling and Loop Tiling

Loop unrolling and loop tiling are two important optimization techniques for improving the performance and efficiency of CNN inference accelerators on FPGAs. These techniques exploit the parallelism and locality of the computation and data access patterns in the convolutional layers, which are the most computationally intensive and time-consuming parts of the CNN inference [154].

Loop unrolling is a technique that reduces the overhead of loop control and increases the parallelism of the computation by replicating the loop body multiple times and adjusting the loop bounds accordingly. In the context of CNN inference accelerators on FPGAs, loop unrolling can be applied to the loops that iterate over the input and output channels, the kernel dimensions, and the output feature map dimensions. For example, consider a convolutional layer with a 3x3 kernel, 64 input channels, and 128 output channels. The computation of this layer can be expressed as a nested loop with six levels: output channel, input channel, output row, output column, kernel row, and kernel column [163]. By unrolling the loops over the output and input channels, the accelerator can compute multiple output and input channels in parallel, increasing the throughput and utilization of the hardware resources [164]. However, the degree of loop unrolling is limited by the available hardware resources on the FPGA, such as the number of DSP slices, memory blocks, and registers. Excessive loop unrolling may lead to resource contention and routing congestion, which can degrade the performance and power efficiency of the accelerator.

Several studies have proposed various techniques for loop unrolling in CNN inference accelerators on FPGAs. For example, Ma et al. [165], the authors proposed a tool called "ALAMO" that automatically generates and optimizes the accelerator design based on the loop unrolling parameters specified by the user. They demonstrated that their tool can achieve up to 4.8x speedup and 3.2x energy efficiency improvement compared to a baseline design without loop unrolling. Similarly, T et at. [166] proposed a technique called "fine-grained dynamic-precision data quantization" that dynamically adjusts the precision of the data based on the range of the activations in each unrolled loop iteration. They showed that their technique can reduce the accelerator’s memory bandwidth and power consumption by up to 50% while maintaining the accuracy of the CNN model within 1% of the full-precision baseline on Zynq FPGA.

On the other hand, loop tiling is a technique that improves the locality and reuse of the data by partitioning the loop iteration space into smaller blocks or tiles and reordering the loop nests to minimize the data movement between the memory hierarchies [167]. It can be applied to the loops that iterate over the input and output feature map dimensions, the kernel dimensions, and the batch size [168]. For example, consider a convolutional layer with a 224x224 input feature map, a 3x3 kernel, 64 input channels, and 128 output channels. The computation of this layer can be tiled into smaller blocks, such as 56x56 input tiles, 64x64 output tiles, and 16x16 kernel tiles. By processing the tiles in a specific order, such as input-channel-first or output-channel-first, the accelerator can maximize the reuse of the input and output data in the on-chip memory and minimize the data transfer between the off-chip memory and the PL part. However, the optimal tile size and tiling order depend on the memory hierarchy and bandwidth of the FPGA platform, as well as the data access patterns and dependencies of the CNN model. Small tiles may lead to frequent data transfers and low reuse, while large tiles may exceed the capacity of the on-chip memory and cause pipeline stalls. Therefore, the design of efficient loop tiling requires careful analysis and optimization of the data flow and memory management of the accelerator [169].

Despite the benefits of loop unrolling and loop tiling for CNN inference accelerators on FPGAs, there are several challenges and opportunities for future research and development in this area. One challenge is the scalability and flexibility of the accelerator design with respect to the CNN model and the FPGA platform. As the complexity and diversity of the CNN models continue to grow and the FPGA platforms continue to evolve, the accelerator design needs to be modular, configurable, and adaptable to different requirements and constraints [170]. Therefore, there is a need for HLS programming models that can abstract the details of the hardware and the software and enable the rapid development and deployment of the accelerator [34]. An opportunity in this area is the integration and acceleration of the pre-processing and post-processing stages of the CNN inference pipeline on the FPGA [39]. These stages, such as image resizing, normalization, and non-maximum suppression, can account for a significant portion of the end-to-end latency and energy consumption of the inference. By offloading these stages to the PL part and optimizing them together with the convolutional layers, it is possible to achieve further speedup and efficiency gains.

Systolic Array Architecture

Systolic arrays are a type of spatial architecture that has been widely used for accelerating CNN inference on SoC-FPGAs. A systolic array consists of a grid of PEs that perform computations in a synchronized and pipelined manner [171]. Each PE is connected to its neighboring PEs in a regular and structured way, allowing data to flow through the array in a rhythmic and parallel fashion [29]. The PEs are typically arranged in a 2D grid where each PE is responsible for performing a single MAC operation [172] and passing the results to their neighboring PEs, which accumulate the partial sums and produce the output feature maps. The input feature maps and weights are stored in on-chip memory BRAMs and are streamed into the array in a choreographed manner. This architecture is particularly well-suited for the matrix-matrix and matrix-vector operations as it can exploit the inherent parallelism and data reuse in these operations [173]. One of the key advantages of systolic arrays is their ability to achieve high throughput and energy efficiency by exploiting the spatial and temporal locality of data [38].

However, designing efficient systolic arrays for CNN inference SoC-FPGAs also presents several challenges. One challenge is the limited on-chip memory capacity, which can limit the size of the CNN models that can be accelerated. To address this challenge, designers can use techniques such as tiling and data reuse to minimize the amount of data movement between the BRAM and DRAM. This has led to successful CNN inference accelerators in a variety of applications, such as image classification, object detection, and semantic segmentation. To implement systolic arrays on SoC-FPGAs, researchers have explored the AMD Vitis-HLS tool, which allows designers to describe the architecture using HLS programming languages (C/C++), which are then automatically translated into HDL code [174]. This abstraction enables faster design iterations, parameterization, and optimization of the systolic array architecture.

Several recent works have demonstrated the effectiveness of systolic arrays for CNN inference acceleration on SoC-FPGAs. For example, Chua et al. [171] proposed a systolic array-based accelerator for quantized CNNs on the ZYNQ chip, achieving a peak performance of 10.98 GOPS and an energy efficiency of 30.26 GOPS/W. Similarly, Zhang et al. [160] proposed a systolic array CNN accelerator for AMD Zynq-7000. The accelerator consists of a 16x16 array of PEs, each capable of performing a MAC operation on an 8-bit input and an 8-bit weight. The array is connected to a set of BRAMs that store the input feature maps, weights, and output feature maps. They demonstrated that the accelerator achieved a peak performance of 200 GOPS at a clock frequency of 200 MHz while consuming only 3.5 W of power. Those accelerators employed a novel output-stationary weight and incorporated techniques such as loop unrolling and pipelining to optimize resource utilization and throughput. In conclusion, systolic arrays have proven to be a highly effective spatial architecture for accelerating CNN inference by exploiting the parallelism, data reuse, and heterogeneous computing capabilities of embedded systems.

Dataflow Architecture

Dataflow architectures have emerged as a promising temporal architecture for accelerating CNN inference on FPGAs. Unlike spatial architectures like systolic arrays, which focus on the spatial distribution of PE, dataflow architectures emphasize the temporal scheduling and orchestration of data movement and computation [36] to maximize data reuse, minimize memory accesses, and improve overall performance and energy efficiency [33]. It is modeled as a directed acyclic graph (DAG), where nodes represent computational tasks and edges represent data dependencies [163] in a pipelined manner, with each node processing the data as soon as it becomes available and passing the results to the next node. Then, the accelerator can exploit the inherent parallelism in the CNN computation, enabling efficient utilization of the FPGA resources. One of the key advantages of dataflow architectures is the ability to leverage the flexibility and reconfigurability of the FPGA fabric to create custom data paths and memory hierarchies to create a specialized pipeline that matches the computational requirements of the CNN. This allows for fine-grained control over the data movement and computation, enabling the optimization of data locality, bandwidth utilization, and memory access patterns [175].

There are several popular dataflow patterns that have been used for CNN inference on FPGAs, such as:

1. Output stationary (OS): In this dataflow, the output feature maps are stored in the on-chip memory of the FPGA, and the input feature maps and weights are streamed through the PEs. This allows for efficient accumulation of the partial sums and minimizes the memory bandwidth required for writing the output data [176].
2. Weight stationary (WS): In this dataflow, the weights are stored in the on-chip memory of the FPGA, and the input feature maps are streamed through the PEs [48]. This allows for efficient reuse of the weights and minimizes the memory bandwidth required for reading the weight data.

3. No local reuse (NLR): In this dataflow, both the input feature maps and weights are streamed through the PEs without any local storage or reuse. This minimizes the on-chip memory requirements but may increase the memory bandwidth and energy consumption deep compress[6].
4. Row stationary (RS): In this dataflow, a row of the input feature maps and a row of the weights are stored in the on-chip memory of the FPGA, and the computation is performed in a systolic manner. This allows for efficient reuse of both the input and weight data and minimizes memory bandwidth and energy consumption [177].

The choice of dataflow pattern depends on the specific characteristics of the CNN model, such as the layer dimensions, kernel sizes, and data precision [178]. It also depends on the available hardware resources on the FPGA, such as the number and size of PEs, memory blocks, and interconnects. Therefore, designing efficient dataflow architectures requires careful analysis and optimization of the data dependencies, resource utilization, and performance trade-offs.

Several recent works have demonstrated the effectiveness of dataflow architectures for CNN inference acceleration on ZYNQ. For example, Umuroglu et al. [48] proposed a dataflow architecture called "FINN" for accelerating CNN inference on Zynq FPGA. The architecture consists of a set of compute engines (CEs) that are connected in a pipeline, with each CE implementing a specific layer of the CNN. The data is streamed through the pipeline in a row-stationary dataflow, with the input and weight data stored in the on-chip memory of the CEs. This architecture achieved a latency of 283 us with CIFAR-10 at a clock frequency of 200 MHz while consuming only 25 W of power. This example demonstrated the potential of dataflow architectures as temporal architectures for accelerating CNN inference that achieved high performance and energy efficiency for a wide range of CNN models and applications. However, designing efficient dataflow architectures requires careful consideration of the data dependencies, resource constraints, and performance trade-offs. As the demand for real-time, low-power, and high-performance CNN inference continues to grow, dataflow architectures on ZYNQ SoC-FPGAs will likely play a crucial role in enabling the deployment of CNNs on edge devices and embedded systems.

Parallel Abstraction of HLS

HLS allows designers to describe the functionality of the accelerator using high-level programming languages, such as C/C++ or OpenCL, which are then automatically translated into HDLs like VHDL or Verilog [176]. This abstraction enables rapid prototyping, design space exploration, and optimization of CNN accelerators without the need for time-consuming and error-prone manual HDL coding. One of the key challenges in designing efficient CNN accelerators on FPGAs is exploiting the inherent parallelism in CNN computation, such as data-level parallelism (DLP) and task-level parallelism (TLP). HLS addresses this challenge by providing a compiler that automatically generates the corresponding HDL code that implements the specified parallelism and optimizations [175].

One common abstraction for expressing parallelism in HLS is the use of parallel loops or loop unrolling [163]. By annotating loops with HLS directives, such as #pragma HLS unroll in AMD Vitis HLS for FPGAs, designers can instruct the HLS compiler to unroll the loop and generate parallel hardware resources for each iteration. This enables the simultaneous processing of multiple data elements, such as input features or output channels, thereby increasing the throughput of the accelerator [179]. Another abstraction for expressing parallelism in HLS is the use of pipelining, which allows the overlapping of different operations in the accelerator, enabling the concurrent execution of multiple CNN layers. HLS directives, such as #pragma HLS pipeline in AMD Vitis HLS can be used to specify the desired pipelining strategy and optimize the hardware architecture for maximum throughput [154]. HLS also provides abstractions for data partitioning and memory optimization, such as #pragma HLS array_partition or #pragma HLS allocation, which can specify how the input data and intermediate results are partitioned and stored in the on-chip memory BRAM. It can also split the 2D array into two smaller arrays, allowing the inner loop to access two elements of A concurrently. This enables the efficient utilization of memory bandwidth and the optimization of data reuse, which are critical for achieving high-performance CNN inference on FPGAs. Additionally, HLS tools provide various pragma directives and libraries to express and optimize task-level pipelining alongside memory hierarchy and access patterns [180], such as:

- The #pragma HLS dataflow directive enables task-level pipelining by creating separate hardware modules for each task and connecting them with FIFOs or streams.
- The hls::stream library provides a set of templated classes for modeling and synthesizing streaming interfaces between tasks, enabling efficient communication and synchronization.
- The #pragma HLS interface directive specifies the type and behavior of the input and output interfaces of a task, such as the handshake protocol, the data width, and the buffer depth.

- The #pragma HLS array_reshape directive changes the shape and layout of an array to improve the memory access efficiency, such as merging multiple small arrays into a larger one, or splitting a large array into multiple smaller ones.
- The #pragma HLS dependence directive provides hints to the HLS tool about the data dependencies and access patterns in the algorithm, enabling more aggressive optimization and parallelization.

Then, the hardware architecture will have a distributed and hierarchical memory system, where the data is stored and accessed in a way that maximizes the memory bandwidth utilization and minimizes the memory access latency.

In recent years, several parallel abstraction frameworks have been proposed to further facilitate the development and optimization of CNN accelerators in HLS, several parallel abstraction frameworks have been proposed in recent years. These frameworks provide high-level APIs, libraries, and tools for expressing and optimizing the parallelism, memory hierarchy, and dataflow of CNN accelerators, hiding the low-level details of HLS and FPGA programming from the user. One such framework is FINN [48], which is an open-source framework for building fast and scalable CNN accelerators on FPGAs. FINN provides a high-level API for specifying the CNN model and the accelerator architecture as well as a set of tools for automatically generating the HLS code and the FPGA bitstream. FINN supports various parallelization and optimization techniques, such as quantization, pipelining, and dataflow, and can achieve up to 200 TOPs/s performance on a single FPGA. Another framework is HLS4ML [49], which converts a trained CNN model into an optimized HLS code for FPGA implementation. The tool uses a DSL to describe the CNN model and the hardware architecture and applies various parallelization and quantization techniques to optimize the HLS code. The authors demonstrated that their tool can achieve up to 9.37 TOPs/s performance and 5.21 TOPs/W efficiency on an AMD Virtex UltraScale+ VU9P FPGA for a range of CNN models and datasets. These frameworks demonstrate the potential of parallel abstraction in HLS for enabling the rapid and efficient development of CNN accelerators on FPGAs. By providing high-level abstractions and automated tools for parallelization, memory optimization, and hardware generation, these frameworks can significantly reduce the development time and effort and improve the performance and efficiency of the resulting accelerators.

Hardware Implementation and Experimental Setup

The hardware implementation involves several critical steps and considerations. AMD SoC-FPGAs, such as the Zynq UltraScale+ series, offer a powerful and flexible platform for implementing CNN accelerators due to their integrated PL and ARM PS [1]. To ensure clarity and precision in describing the hardware implementation and experimental setup for CNN accelerators, we can organize the process into detailed sequential steps using tools like Vitis HLS, Vivado, and PYNQ at each stage:

A. High-Level Synthesis with Vitis HLS.

   1. FPGA Platform Selection and Setup:

      - Select an appropriate AMD SoC-FPGA platform based on the requirements of the CNN inference accelerator, considering factors such as computational complexity, performance metrics, power and resource constraints, and cost [1].

- Resource Utilization: Estimate the FPGA resources (DSPs, BRAMs, LUTs) needed to achieve the performance targets without exhausting the FPGA's capabilities.

2. Develop HLS IP for CNN using Vitis HLS:
    - Create a new Vitis HLS project and specify the target FPGA platform and clock frequency.
    - Implement the CNN accelerator architecture using HLS in C/C++.
    - Optimize the HLS code for performance and resource efficiency by applying techniques such as loop unrolling, pipelining, and dataflow optimization.
    - Specify the interface protocol (e.g., AXI4, AXI4-Stream) for the accelerator's input/output ports.
    - Simulate and validate the functionality of the HLS code using test benches and reference models.
    - Synthesize the HLS code into register-transfer level (RTL) code (e.g., VHDL, Verilog) and generate the IP core.
3. Optimization and Iteration:
    - Feedback Loop: Iterate on the design by adjusting the HLS pragmas based on synthesis results to better meet or balance performance objectives.

B. Detailed FPGA Implementation with Vivado:

1. Import RTL into Vivado and configure the block design:
    - Create a new Vivado project and specify the target FPGA platform and design constraints.
    - Import the generated IP core from Vivado HLS into the Vivado project.

- Integrate the CNN accelerator IP with other system components, such as the Zynq PS, memory controllers, and communication interfaces, using the Vivado IP Integrator.
- Configure the PS-PL interface (e.g., AXI4, AXI4-Lite) and memory mapping for efficient data transfer between the PS and the accelerator.

2. Set Up FPGA Constraints:
   - Timing Constraints: Define the clock frequencies and the timing requirements for interfaces.
   - I/O Pin Assignments: Specify the mapping of input/output pins on the FPGA to facilitate connections to other hardware or peripherals.
3. Implementation Tools:
   - Place and Route: Execute the placement and routing processes. This step involves the physical allocation of the logic to specific locations on the FPGA and the planning of interconnects between them.
   - Timing Analysis: Conduct extensive timing analysis to ensure all paths meet timing requirements, adjusting placement and routing as necessary.
   - Generate Bitstream: Produce the bitstream file, which contains all the programming data for the FPGA.
4. Hardware Validation:
   - Real Hardware Testing: Program the FPGA with the bitstream and test it in a real-world scenario to verify it meets the functional and performance specifications.

- Generate the bitstream for FPGA configuration and export the hardware platform for software development.

C. Experimental Setup with PYNQ:

1. Prepare the PYNQ Environment:
    - Setup and Configuration: Install the PYNQ image on the ZYNQ board, set up the Jupyter Notebook server, and install necessary Python libraries, such as NumPy, OpenCV, and TensorFlow, PyTorch.
2. Deploy CNN Model:
    - Model Conversion and Deployment: Convert the trained CNN model into a format compatible with the PYNQ environment and upload it to the FPGA.
    - Load the hardware platform and the bitstream generated from Vivado into the PYNQ environment.
    - Develop software applications to control and interact with the CNN accelerator using PYNQ's Python APIs and libraries.
3. Run Inference:
    - Data Handling: Process input data through the FPGA-accelerated model, using PYNQ to manage data transfer between Python and the FPGA.
4. Performance Measurement:
    - Metrics Collection: Gather data on key performance indicators. Utilize built-in PYNQ tools to measure throughput, latency, and power consumption.
5. Iterate and Optimize:

- Performance Tuning: Based on the collected metrics, tweak the FPGA design or Python code to optimize performance, reducing bottlenecks or enhancing efficiency.

By following these steps and leveraging the capabilities of Vitis HLS, Vivado, and PYNQ, researchers and developers can effectively implement, evaluate, and deploy CNN inference accelerators on FPGAs. The combination of HLS, IP integration, and software-hardware co-design enables rapid prototyping, performance optimization, and scalable deployment of CNN accelerators for various applications.

Open-Source HW-SW Co-Design Framework

The rapid advancement of CNNs has led to a growing demand for efficient and high-performance hardware accelerators. However, designing and deploying CNN accelerators can be a complex and time-consuming process, requiring expertise in both hardware and software domains. To address this challenge, open-source HW-SW co-design frameworks have emerged as a promising solution, enabling researchers and developers to collaborate, innovate, and accelerate the development of CNN accelerators. These frameworks provide a unified environment for the joint design, optimization, and deployment of CNN accelerators, taking into account both hardware and software considerations. These frameworks typically include a set of tools, libraries, and methodologies that facilitate the exploration of different hardware architectures, software optimizations, and system-level integration. FINN [48] and HLS4ML [49] are two popular open-source HW-SW co-design frameworks that have gained significant attention in the research community for their contributions to the acceleration of CNNs on FPGAs. Let's discuss each of these frameworks in more detail.

FINN Framework

FINN is an open-source framework developed by AMD Research Labs (now AMD) that aims to provide fast, scalable, and flexible CNN accelerator inference on FPGAs [48]. FINN leverages HLS and quantization techniques to reduce the precision of weights and activations, enabling more efficient hardware implementations without significant loss in accuracy. Key features of FINN include:

- Support for various quantization schemes, such as binary, ternary, and fixed-point quantization.
- Automated generation of optimized dataflow architectures based on the quantized CNN model.
- Streamlined design space exploration and performance estimation tools.
- Integration with popular deep learning frameworks, such as TensorFlow and PyTorch, through the ONNX (Open Neural Network Exchange) format.

The generated hardware accelerator leverages the parallelism and fine-grained control offered by FPGAs to achieve high performance and energy efficiency. FINN employs a streaming dataflow architecture, where the computation is organized as a pipeline of PEs that operate on the input data in a synchronized manner. Each PE performs a specific operation, such as convolution, pooling, or activation, and communicates with other PEs through high-bandwidth on-chip connections. This dataflow architecture enables efficient data reuse, minimizes off-chip memory accesses, and allows for high throughput processing of the neural network inference.

FINN has been extensively evaluated on various FPGA platforms and CNN models, demonstrating significant improvements in performance and energy efficiency compared to CPU and GPU implementations [181]. For instance, on the AMD ZC706 FPGA board, FINN achieved a throughput of 12.45 TOP/s (tera operations per second) for the AlexNet CNN model, which is 8.3× higher than the throughput of an NVIDIA Titan X GPU. As a result, it is a powerful and comprehensive framework for fast, scalable, and efficient deployment of deep learning inference on FPGAs. By leveraging the flexibility and performance of FPGAs, along with advanced optimization techniques such as quantization, pruning, and folding, FINN enables the acceleration of CNN models with high throughput and energy efficiency. The ease of use, accessibility, and integration capabilities of FINN make it an attractive choice for researchers and developers seeking to harness the benefits of FPGAs for deep learning inference in various applications. Additionally, it has a strong and active community that contributes to the development of the framework by submitting bug reports, feature requests, and pull requests. The community also engages in discussions and knowledge sharing through forums, mailing lists, and online platforms.

HLS4ML Framework

HLS4ML is an open-source framework that aims to simplify the deployment of machine learning algorithms on FPGAs using HLS and focuses on the acceleration of CNN inference for scientific applications [182]. It is developed by a collaboration between the Fermi National Accelerator Laboratory, CERN, and Stanford University. The toolkit takes a trained neural network model, specified in a high-level framework such as TensorFlow or PyTorch, and converts it into a synthesizable hardware description using HLS. By leveraging HLS, HLS4ML allows machine learning practitioners to focus on the design and training of their models, while automatically handling the low-level hardware implementation details. Key features of HLS4ML include:

- Automated translation of trained CNN models into HLS code (C/C++) for FPGA implementation and other hardware acceleration platforms, such as ASICs and SoCs.
- Support for various CNN architectures and layers, such as convolutional, dense, and activation layers.
- Support newer neural network architectures, such as Graph Neural Networks (GNN) and transformers.
- Configurable precision settings for weights and activations, enabling trade-offs between accuracy and resource usage.
- Integration with popular machine learning frameworks, such as TensorFlow and PyTorch, through model serialization formats like ONNX.

HLS4L leverages the capabilities of modern HLS tools, such as AMD Vitis HLS and Intel OpenCL SDK for FPGAs, to generate optimized hardware designs from high-level descriptions of CNN models. To facilitate the deployment and integration into larger systems, the framework provides a set of driver and interface code for connecting the accelerators to host processors and external memory. This includes C++ and Python APIs for configuring and controlling the accelerators, as well as RTL templates for integrating the accelerators into FPGA-based systems using standard interfaces such as AXI4 and PCIe. The framework also includes a set of tools and scripts for automating the build and deployment process, such as Makefiles and Tcl scripts for Vivado and Quartus.

The HLS4ML framework has been extensively validated and benchmarked on a range of FPGA platforms and devices, including AMD Zynq, Virtex, and Kintex devices, as well as Intel Arria and Stratix devices. The framework has also been successfully applied to various scientific use cases, such as the acceleration of CNN-based trigger systems for particle physics experiments, and the implementation of real-time object detection and classification on FPGAs. Recent advancements in HLS4ML include the incorporation of pruning and quantization techniques for further optimization of CNN models [183]. The HLS4ML project has a growing community of users and contributors, who are actively involved in extending and improving the toolkit. The project maintains an open-source repository on GitHub, which serves as a central hub for development, collaboration, and issue tracking [49].

Optimized HW-SW Co-Design Framework

The demand for deploying CNNs on edge devices has grown significantly due to the increasing popularity of AI-driven applications in various domains, such as autonomous vehicles, smart surveillance, and industrial automation [184]. However, the computational complexity and memory requirements of CNNs pose significant challenges for resource-constrained edge devices, which often have limited processing power, memory, and energy budgets. To address these challenges, several closed-source optimized HW-SW CNN frameworks have been developed by leading technology companies, such as AMD, Intel, and NVIDIA. These frameworks aim to accelerate CNN inference on edge devices by leveraging specialized hardware accelerators, such as FPGAs, ASICs, and GPUs, and applying various optimization techniques to reduce the computational and memory overhead of CNN models. Those frameworks offer a comprehensive solution for deploying CNNs on edge devices, providing a full stack of tools, libraries, and runtime environments to simplify the development and optimization process. They typically include the following key components:

- Model Optimization Tools: These tools help developers optimize CNN models for specific hardware targets by applying techniques such as quantization and pruning to reduce computational complexity while maintaining acceptable accuracy [14].
- Hardware Acceleration Libraries: These libraries provide optimized implementations of CNN operations, such as convolution, pooling, and activation functions, that are tailored for specific hardware accelerators. These libraries take advantage of the parallelism and memory hierarchy of the hardware to maximize performance.
- Compilation and Deployment Tools: These tools automate the process of compiling and deploying optimized CNN models on edge devices. They handle the mapping of CNN operations onto the hardware accelerator, manage memory allocation and data movement, and generate efficient executable code [185].
- Runtime Environment: The runtime environment provides a set of APIs and libraries to enable the execution of optimized CNN models on edge devices. It manages the scheduling of computation and data transfer, handles synchronization and communication between the host processor and the hardware accelerator, and provides performance monitoring and profiling capabilities.

By leveraging these components, developers can accelerate CNN inference on edge devices with minimal manual effort. Some of the most notable closed-source optimized hardware-software CNN frameworks include AMD Vitis-AI [62], Intel oneAPI [186] with OpenVINO [63], and NVIDIA TAO Toolkit [187] with TensorRT [64]. These frameworks have gained significant adoption in industry and academia due to their performance, ease of use, and strong ecosystem support.

AMD Vitis-AI is a comprehensive development platform that enables the deployment of accelerated AI applications on AMD FPGAs and SoCs. It provides a unified software stack, including IP cores, optimized libraries, pre-built models, and tools for quantization, pruning, and compilation to simplify the development process and achieve high performance and efficiency. Vitis AI supports a wide range of CNN models and frameworks, such as TensorFlow and PyTorch and offers runtime support for edge devices like the AMD Zynq UltraScale+ MPSoC [180].

Intel OneAPI is a unified programming model and toolkit that enables developers to write high-performance, cross-architecture applications for various platforms, including CPUs, GPUs, VPUs, and FPGAs [188]. It includes a deep learning toolkit called (oneDNN), which provides optimized building blocks for deep learning frameworks and enables efficient inference on Intel hardware. OneAPI can be integrated with Intel OpenVINO which is another prominent framework that facilitates the optimization and deployment of CNN models on Intel hardware. It provides a set of tools and libraries for model optimization, such as quantization, layer fusion, and post-training optimization, as well as runtime inference engines for various platforms. OpenVINO also integrates with OpenCV for computer vision and image processing tasks for applications ranging from robotics and industrial automation to smart city and retail analytics [189].

NVIDIA TAO Toolkit is a low-code AI toolkit that simplifies the training, optimization, and deployment of AI models on NVIDIA hardware, including GPUs and Jetson edge devices [190]. It offers a set of pre-trained models, transfer learning capabilities, and pruning techniques to optimize models for specific tasks and target platforms]. TAO Toolkit integrates with the NVIDIA TensorRT inference optimizer and runtime, enabling high-performance inference on edge devices with real-time constraints [191].

These closed-source optimized hardware-software CNN frameworks offer end-to-end solutions for accelerating deep learning inference on edge devices. They abstract the complexity of hardware-specific optimizations and provide high-level APIs, tools, and libraries to simplify the development process [185]. However, the closed-source nature of these frameworks may present certain limitations, such as the lack of flexibility and customization options for specific use cases or the inability to integrate with proprietary hardware accelerators. Additionally, the licensing and cost associated with these frameworks may be a consideration for researchers, developers, and organizations. Despite these limitations, they provide a powerful and accessible solution for deploying AI applications on resource-constrained edge devices, enabling new possibilities for intelligent and responsive systems at the edge.

Standardize HW-SW Co-Design Framework

The deployment of deep learning models on edge devices, such as smartphones, IoT devices, and embedded systems, poses significant challenges due to the limited computational resources and power budgets of these devices. To address these challenges, there has been a growing interest in developing efficient and generic methods for representing, optimizing, and deploying DNNs on edge devices. ONNX (Open Neural Network Exchange) [65] and ONNC

(Open Neural Network Compiler) [192] are two prominent open-source projects that have emerged to tackle these challenges. ONNX is an open-source format for representing deep learning models, enabling interoperability between different frameworks and tools. ONNC, on the other hand, is a compiler framework that takes ONNX models as input and generates optimized code for various hardware targets, particularly edge devices.

The development of ONNX was motivated by the fragmentation of the deep learning ecosystem, with various frameworks, such as TensorFlow, PyTorch, and MXNet [193] using different model formats and APIs. This fragmentation made it difficult to exchange models between frameworks and deploy them on different hardware platforms. ONNX addresses this issue by providing a standard way to define the computation graph, operators, and tensors of a neural network, enabling models to be trained in one framework and deployed in another. ONNX has also gained significant adoption in the industry, with support from major technology companies, including Microsoft, Facebook, Amazon, Intel, AMD, and NVIDIA. It has become a crucial component in the deep learning toolchain, enabling the deployment of models on a wide range of platforms, from cloud servers to edge devices.

ONNC on the other hand, was built upon the ONNX format to provide a comprehensive solution for optimizing and deploying deep learning models on edge devices. The main goal of ONNC is to bridge the gap between the high-level representations of deep learning models and the low-level, hardware-specific optimizations required for efficient execution on embedded system. ONNC achieves this goal by employing a multi-stage optimization pipeline that takes an ONNX model as input and applies a series of optimization passes to generate efficient executable code for the target hardware. These optimization passes include operator fusion, constant folding, and memory allocation optimization, which aim to reduce the computation and memory footprint of the model. One of the key strengths of ONNC is its modular and extensible design, which allows for the integration of new optimization techniques and hardware backends. This flexibility enables ONNC to support a wide range of hardware targets, from general-purpose processors CPU to specialized accelerators, such as GPUs, FPGAs, and ASICs.

In conclusion, ONNX and ONNC provide a powerful framework for optimizing and deploying deep learning models on edge devices. The open-source nature and wide industry adoption of ONNX ensure model interoperability and portability, while ONNC's modular architecture and extensive hardware support enable efficient acceleration of deep learning inference on diverse edge platforms. Together, they form a comprehensive ecosystem that supports end-to-end deployment of neural networks, from model training on one platform to efficient execution on another, As the demand for edge AI continues to grow, ONNX and ONNC are expected to play an increasingly important role in enabling the deployment of intelligent and responsive applications on resource-constrained devices.

Discussion

This review provided a comprehensive overview of the fundamentals of deep learning and CNNs and emphasized the importance of model compression and optimization techniques, such as pruning, quantization, and knowledge distillation, along with state-of-the-art CNN architectures and datasets commonly used for benchmarking and evaluation. We demonstrated an end-to-end toolchain for training, optimizing, and deploying CNNs on edge FPGA using AMD Vitis HLS and Vivado environments alongside others framework. In particular, we delved into the HW-SW co-design approaches for accelerating CNN inference on edge devices with a focus on the accelerator design, including the PS, PL, PEs, SIMD, memory hierarchy, and data flow optimization techniques using HLS. The most difficult part of this research was the emerging trends and opportunities in the field of CNN acceleration, such as the use of open-source frameworks like FINN and HLS4ML, as well as the adoption of standardized representations like ONNX and ONNC for improved interoperability and portability of CNN models However, the review also acknowledges the challenges and limitations associated with deploying CNNs on edge devices, such as the trade-offs between accuracy, performance, and resource utilization, as well as the complexity and learning curve associated with hardware-software co-design and FPGA programming.

CHAPTER THREE: GENERATIVE ADVERSARIAL NETWORK ON THE EDGE

Introduction

Generative Adversarial Networks (GANs) [54] have emerged as a robust generative modeling algorithm capable of producing realistic samples across various domains, including images, videos, and audio. GANs consist of two neural networks, a generator G and a discriminator D, that engage in a competitive game. The generator aims to create realistic samples that resemble the data distribution, while the discriminator tries to distinguish between actual and generated samples, as shown in Fig. 22.

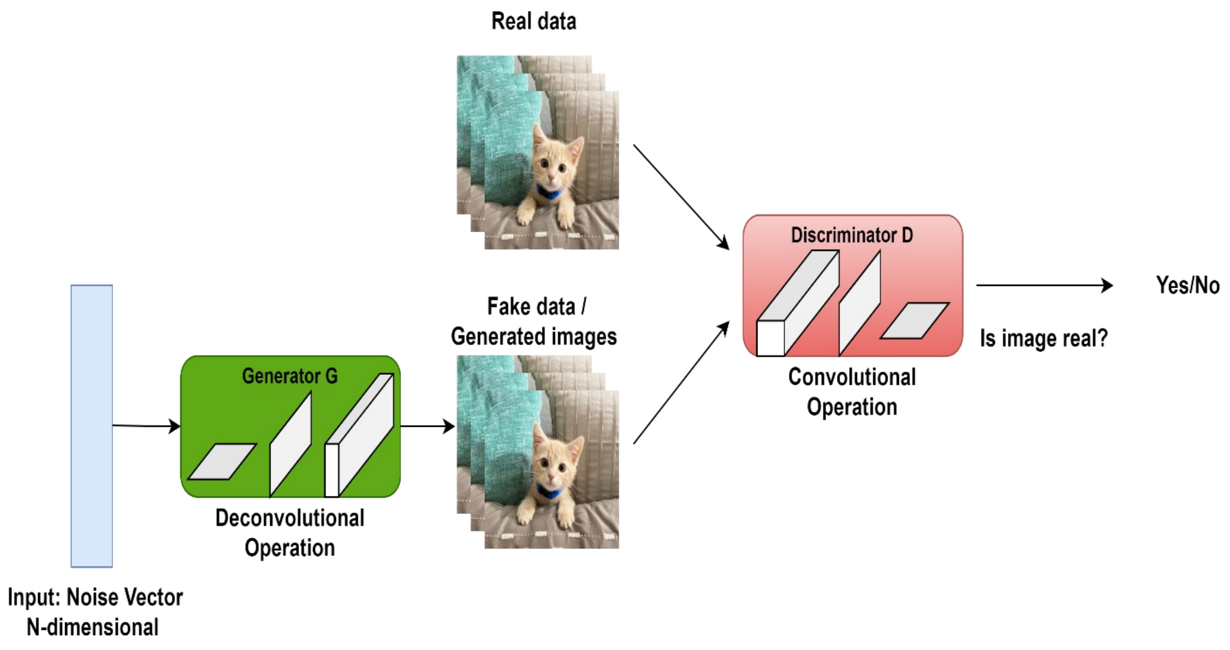

Figure 22. Generative Adversarial Networks working principle

The generator's objective is to maximize the probability of the discriminator making a mistake, while the discriminator aims to minimize the classification error. This adversarial training process can be formulated as a minimax game, where the generator and discriminator are optimized alternately [194]. The generator inputs a random noise vector z and maps it to the data space through convolutional, upsampling, and deconvolutional layers [195]. On the other hand, the discriminator takes both the natural and generated sample as input and outputs a probability score indicating the likelihood of the sample being real [196]. The architectures of the generator and discriminator can vary depending on the specific application and the type of data being modeled [197].

During training, the generator and discriminator are updated iteratively using backpropagation and gradient-based optimization techniques, such as stochastic gradient descent (SGD) or Adam. The generator is trained to minimize the adversarial loss, encourages it to produce samples that fool the discriminator. The discriminator, in return, is trained to maximize the log-likelihood of correctly classifying actual and generated samples [198]. One of the main challenges in training GANs is ensuring stability and convergence [199]. The training process can be sensitive to hyperparameter settings, network architectures, and optimization techniques [200]. Mode collapse, where the generator produces a limited variety of samples, and vanishing gradients are common issues hindering the learning process. Various techniques have been proposed to address these challenges, including modified objective functions, regularization methods, architectural improvements, and Wasserstein GANs (WGANs) [196], which use the Wasserstein distance to stabilize the objective function.

Another challenge in GANs is the evaluation and comparison of generated samples. Traditional metrics, such as log-likelihood or perplexity, are not directly applicable to GANs, as the generator does not explicitly estimate a probability distribution [201]. Instead, researchers have proposed alternative evaluation measures, such as the Inception Score (IS) [194] and the Fréchet Inception Distance (FID) [202], which assess the quality and diversity of generated samples based on pre-trained classifiers. The IS measures the quality and diversity of generated samples by using a pre-trained Inception network. It calculates the expected Kullback-Leibler divergence between the conditional class distribution predicted by the Inception network and the marginal class distribution. Higher IS values indicate better quality and diversity of generated samples.

On the other hand, The FID compares the distributions of generated and actual samples in the feature space of a pre-trained Inception network. It calculates the Fréchet distance between the two distributions, which considers the features' mean and covariance. Lower FID values indicate better similarity between the generated and actual distributions. In Table 3.1, Wang et al. [203] summarized the performance of GANs using IS and FID for four different datasets, which are the most widely used as a benchmarking dataset.

Lastly, GANs have undergone various architectural modifications to improve performance and address specific challenges. The original GAN architecture struggled with generating high-quality images and suffered from training instability. Subsequent architectures, such as Deep Convolutional GANs (DCGANs) [195], introduced convolutional layers and batch normalization, significantly improving the quality of generated images and training stability. Further architectural advancements include using Self-attention mechanisms GANs (SAGAN) [204], which enable long-range dependencies and improve the global coherence of generated samples. Additionally, the style-based generator architecture proposed in StyleGANs [197] separates the high-level attributes from the low-level details, allowing for more control over the generated images and enabling the synthesis of highly realistic and diverse samples.

Table 2. Performance summary across different types of GANs discussed in Wang's paper on different datasets. "-" to experiments that have not been done in the literature.

Model	CIFAR10 (IS/FID)	ImageNet (IS/FID)	LSUN (FID)	CelebA (FID)
FCGAN	6.41 / 42.6	- / -	-	-
BEGAN	5.62 / -	- / -	-	83.3
PROGAN	8.80 / -	- / -	8.3	7.3
LSGAN	6.76 / 29.5	- / -	216.	-
DCGAN	6.69 / 42.5	- / 74.2	16.1	63.1
WGAN-GP	8.21 / 21.5	11.6 /62.1	22.8	-
SNGAN	8.43 / 18.8	36.8 / 27.6	-	-
Geometric GAN	- / 27.1	- / -	-	-
RGAN	- / 15.9	- / -	-	-
AC-GAN	8.25 / -	- / -	-	-
BigGAN	9.22 / 14.7	166.5/ 7.4	-	-
RealnessGAN	- / 34.6	- / -	-	23.5
MSG-GAN	- / -	- / -	5.2	8.0
SS-GAN	- / 15.7	- / 43.9	13.3	24.36
YLG	- / -	57.2 / 15.9	-	-
Sphere GAN	- / -	- / -	16.9	-

GAN Challenges in Edge Computing

The deployment of GANs on edge devices has gained significant attention in recent years due to the increasing demand for real-time and privacy-preserving applications. Edge devices, such as smartphones, IoT devices, and embedded systems, often have limited computational resources and power constraints [205]. However, the ability to generate realistic samples directly on edge devices opens up new possibilities for applications like augmented reality, personalized content creation, and anomaly detection. One of the key challenges in deploying GANs on edge devices is the computational complexity of the models. GANs typically require deep neural network architectures and iterative training processes, which can be resource-intensive and time-consuming. To address this challenge, various techniques have been proposed to optimize GAN models for edge deployment, including model compression and architecture simplification [206].

Hardware acceleration is another crucial aspect of enabling efficient GAN deployment on edge devices [207]. Specialized hardware accelerators, such as GPUs, FPGAs, and ASICs, can significantly speed up the computation of GAN models by exploiting parallelism and optimizing memory access patterns [208]. GPU acceleration has been widely used for GAN training and inference, leveraging the massively parallel processing capabilities of GPUs. FPGAs offer flexibility and energy efficiency, allowing for custom hardware designs tailored to the specific requirements of GAN models. ASICs provide the highest performance and energy efficiency but require significant development efforts and upfront costs.

Related Work

Benchmarking studies have also been conducted to evaluate the performance and efficiency of GAN models on edge devices. In [209], Shrivastava et al. presented a comprehensive survey of hardware acceleration techniques for GANs and conducted benchmarking experiments on various edge devices, including NVIDIA Jetson boards, FPGA, and Intel Neural Compute Sticks. They evaluated the inference time, power consumption, and generation quality of different GAN architectures and optimization techniques. Their results showed that model compression techniques can significantly reduce the inference time and memory footprint of GAN models with minimal impact on generation quality. For example, a compressed DCGAN model achieved a 4.2x speedup and a 3.8x reduction in memory usage compared to the original model, with only a slight degradation in the IS from 4.5 to 4.2.

In [203], Wang et al. conducted a benchmarking study of GAN deployment on mobile devices, comparing the performance of different GAN architectures and acceleration frameworks. They evaluated the inference time, power consumption, and FID of various GAN models on an NVIDIA Jetson TX2. Their results demonstrated that MobileNet-based GAN architectures and TensorFlow Lite acceleration framework provide the best trade-off between performance and efficiency for mobile deployment. A MobileNet-based GAN model achieved an FID of 26.5 and an inference time of 18.2 ms on the NVIDIA Jetson TX2, compared to an FID of 29.3 and an inference time of 45.6 ms for a standard DCGAN model.

Yazdanbakhsh et al. [210] proposed a hardware-efficient GAN architecture, called FlexiGAN, which is optimized for FPGA deployment. They designed a lightweight generator network using depth-wise separable convolutions and a resource-efficient discriminator network using binarized weights. FlexiGAN achieved a peak performance of 158.7 GFLOPS and an energy efficiency of 14.2 GFLOPS/W on a AMD ZCU102 FPGA while consuming only 1.8 MB of on-chip memory. Compared to a GPU-based implementation, FlexiGAN demonstrated a 7.5x speedup and a 5.2x energy efficiency improvement, with comparable generation quality.

Objective

Deconvolution is the spatial inverse of convolution as it transforms the input to a higher dimension of output, as shown in the proposed architecture in Fig. 3.2. When applied to photos, it results in an up-sampling. While most open-source projects focus on CNN, which employs an ordinary convolution, DCGAN has received less attention regarding the implementation on edge platforms than convolution and CNN. Even so, deconvolution is used in image super-resolution GAN SRGAN [211] for mapping a low-resolution picture to a higher-resolution image. This is specially used in microscopic images [212] to obtain high-quality photos and produce results in real-time. An efficient implementation of deconvolution also enables the deployment of super-resolution on low-power edge devices and opens the possibility of having technologies like NVIDIA DLSS on low-power edge computing.

The most recent work that approaches our idea is proposed by Colbert et al. [213], which implemented a deconvolution CNN (DCNN) in the PYNQ-Z2 device. Nevertheless, this research is qualitatively distinct from the past one. Several issues are introduced and solved in this paper. To begin with, it is constrained to a 32-bit fixed-point implementation without the ability to experiment with other bit widths. Second, the accelerator is a non-scalable systolic array, which implies that each layer is executed sequentially by a single engine, resulting in increased latency. Third, the weight of each layer is streamed from the DDR (there is no on-chip storage for all the network weights), resulting in increased power consumption. Lastly, it does not make the quantized training, accelerator design, and deployment code open-source to enable experimentation with different network sizes and accelerator design choices.

This study is focusing on utilizing FINN by AMD [48], which is an open-source research framework for accelerating DNNs inference on FPGAs. They provide efficient building blocks for training the quantized neural network (QNN) and developing inference accelerators while only supporting limited DNN layers. In this paper, FINN was extended to train a quantized deconvolutional GAN (DCGAN) [214], designed a scalable accelerator, and deployed the inference on SoC-FPGAs. We also benchmarked our implementation against NIVIDA Jetson Nano and achieved a superior throughput-to-power ratio when running the inference.

In this study, we proposed an HW/SW co-design approach for training quantized deconvolution GAN (QDCGAN) implemented on FPGA using a scalable streaming dataflow architecture capable of achieving higher throughput versus resource utilization trade-off. The developed accelerator is based on an efficient deconvolution engine that offers high parallelism with respect to scaling factors for GAN-based edge computing. Furthermore, various precisions, datasets, and network scalability were analyzed for low-power inference on resource-constrained platforms. Lastly, an end-to-end open-source framework is provided for training, implementation, state-space exploration, and scaling the inference using Vivado HLS for AMD SoC-FPGAs and a comparison testbed with Jetson Nano. The contributions of this research are as follows:

- Developed a scalable inference accelerator for transpose convolution operation for quantized DCGAN (QDCGAN) on top of FINN.
- Provided a complete open-source framework [214] (training to implementation stack) for investigating the effect of variable bit widths for weights and activations.
- Demonstrated that the weights and activations influence performance measurement, resource utilization, throughput, and the quality of the generated images.

Methodology

Network Architecture

Training GAN comes in a zero-sum game between two competing networks: the generator G and the discriminator D. G tries to maximize the loss of D by mapping a noise vector to the input space while D objectives to maximize the chance to identify the real distribution of data. Basically, G is trained to fool D as in the equation as follows:

$$min_G max_D \; E_{x \sim p_r}[log\; D(x)] + E_{\tilde{x} \sim p_g}\left[\log\left(1 - D(\tilde{x})\right)\right] \tag{3.1}$$

Where Pr is the data distribution, and Pg is the model distribution implied by (the input z to the generator is sampled from the noise distribution p). Since GAN are inherently unstable, Wasserstein GAN loss (WGAN) along with the gradient penalty proposed by Gulrajani et al. [215] is applied to stabilize the training and optimize G. The WGAN loss function equation is as follows:

$$min_G max_D \; E_{x \sim p_r}[\, D(x)] + E_{\tilde{x} \sim p_g}[D(\tilde{x})] \tag{3.2}$$

Adding the gradient penalty term as follows:

$$L = E_{\tilde{x} \sim p_g}\left[D(\tilde{x}) - \{E_{x \sim p_r} * D(x)\} + \lambda * E_{\tilde{x} \sim P_{\hat{x}}}\{(\|\nabla_{\tilde{x}} D(\hat{x})\|_2 - 1)^2\}\right] \tag{3.3}$$

D works as a continuous function and is not trained to classify the data, represent a straight line between the data distribution of Pr & Pg with a fixed coefficient that has been proven to work on various architectures and datasets. Besides, applying the previous critical model eliminates the need for batch normalization [152] in any network, as well as calculating different thresholds for every layer and only using the simple thresholds (quantize ReLU), which resulted in better image quality. In contrast, weights and activations were quantized in the forward pass using a hard tanh function followed by an n-bit width quantizer. The following equation is for rounding and clipping the weights:

$$Q(x,n) = Clip\left\{\frac{round(x*2^n)}{2^n}, -1, 1\right\} \tag{3.4}$$

x is the input, whereas n represents the number of bits. If the magnitude of the input during the forward pass is $1 < x \leq -1$, the straight-through estimator (hard tanh) will pass the gradients. Yet, if the input is outside this range, hard tanh will zero out the gradients according to the following equation:

$$g_x = \{g_q * 1_{|x| \leq 1}\} \tag{3.5}$$

Prior equations were applied to improve and stabilize the training of QDCGAN on MNIST and celebA datasets. These publicly available datasets were the most commonly used for new GANs. Besides, the developed QDCGAN architectures shown in Fig. 23 were trained in full-precision weight (W) and activation (A), e.g., W32A32, and then fine-tuned to lower bit-widths (fixed point), e.g., W4A4 and W1A2.

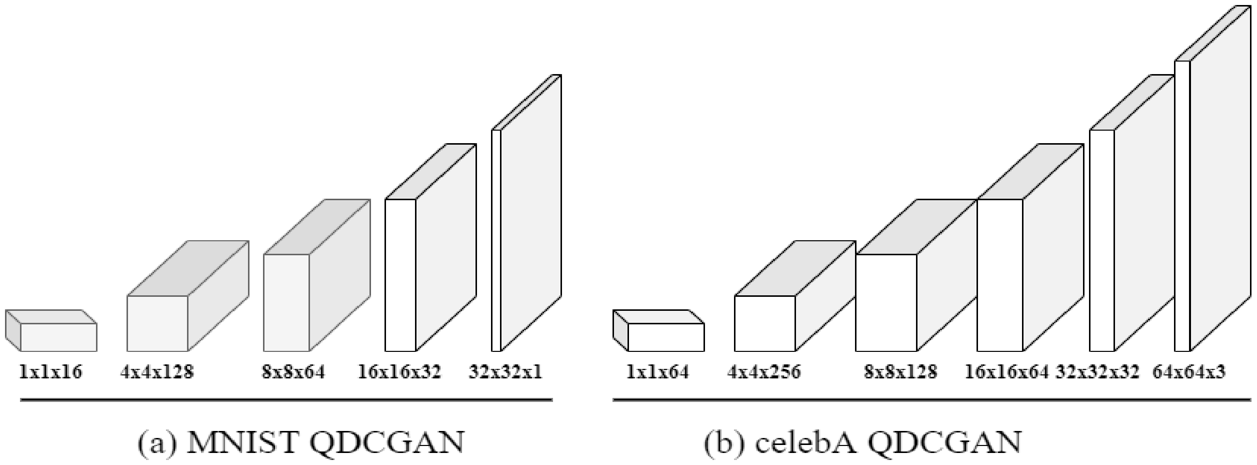

Figure 23. QDCGAN architectures for training and inference acceleration on FPGA.

Accelerator Design

The accelerator was built on top of FINN, and it uses the processing element (PE) and the single instruction multiple data (SIMD) scaling factor for each layer separately. Every layer has its engine for execution. The resources utilization versus throughputs is balanced by variable precisions and numbers of PE & SIMD, which define the inputs and outputs parallelism respectively on the PL part as shown in Fig 24.

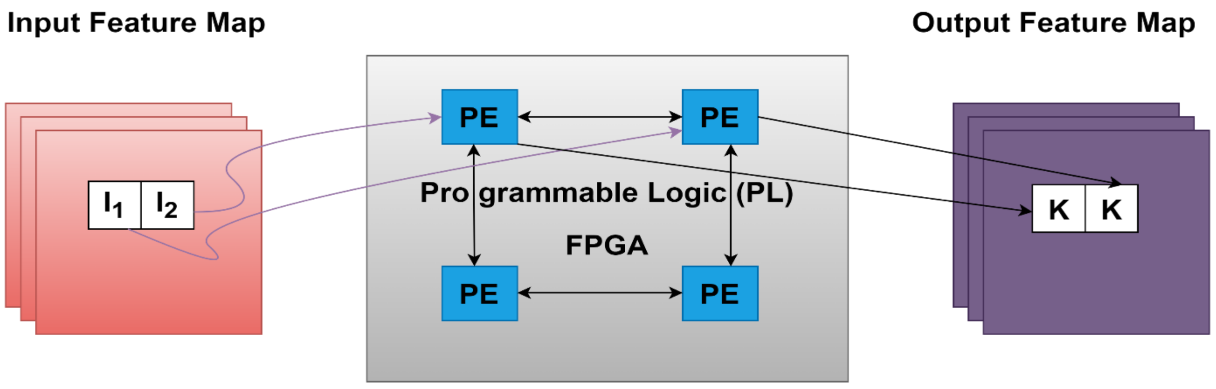

Figure 24. I_1 to I… are neighboring activations of the input feature map. They are mapped to neighboring PEs, where they are multiplied with a kernel K and produce an output feature map result. These results are aggregated over time.

Moreover, the overall parallelism is calculated by the folding factor (FF) as in the following equation:

$$FF = \left\{\frac{H*W}{PE*SIMD}\right\} \tag{3.6}$$

H and W represent the height and width of the matrix, respectively. The smaller the folding factor, the lower the latency to execute the weight matrix. Moreover, the developed accelerator uses a transpose convolution that works by applying certain degrees of expansion and padding of zeroes between the input feature map values. The expansion and padding values are determined by the stride and filter size of the deconvolution layer. Using this pre-processing step of learnable up-sampling, deconvolution can be implemented with an efficient convolution accelerator. Fig. 25 demonstrates an example of how this method helps in the implementation of GAN-based edge devices. The sliding window generator for deconvolution uses a circular ring buffer, which has an efficient mechanism for maintaining and moving a list of values systematically. It helps to store a small portion of the input feature map channels on-chip.

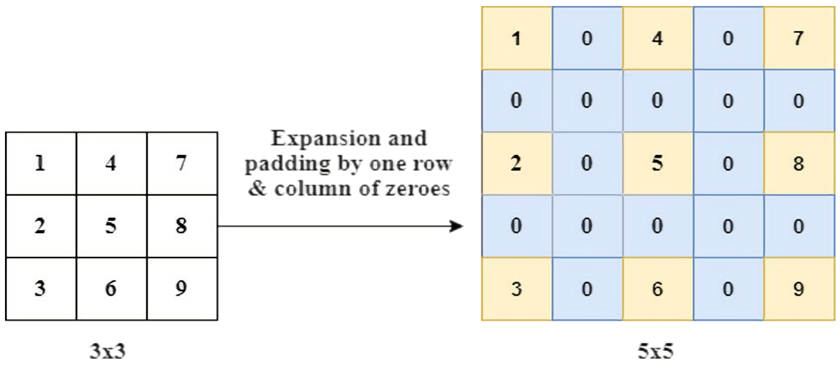

Figure 25. Pre-processing of up-sampling (expansion) for implementing deconvolution operation as a convolution.

Subsequently, the accelerator receives inputs from the processor's DDR, executes the layers, and outputs the results back to the processor. The weights are then stored on-chip (BRAM) and can be accessed concurrently by the partitioned PE elements in parallel, which results in fast execution. After training, the generator network's weights are then extracted, packed according to the PE and SIMD of each layer, and saved in an FPGA-readable format. Hence, the proposed accelerator is then introduced in the FINN-HLSLIB open-source library [216], synthesized to run at 125 MHz, and implemented on SoC-FPGAs. To this end, the network architecture is customizable for larger platforms and datasets.

Host Code with PYNQ

Python with PYNQ API [35] is used to write the driver code that communicates with the hardware, load the accelerator bitstreams into the programmable-logic (PL), loads the weights, passes input/output buffers from DDR, and convert the output data into a visual format to the user.

Experimental Setup and Results

Development Environment

Nvidia Tesla P4 GPU was used to train the QDCGAN. PyTorch was used as a training framework, while Brevitas library from AMD [217] was used for quantization aware-training. Brevitas is a flexible quantization library utilized for training quantized DNNs and deconvolution as well. It is also capable of adjusting and controlling the bit width of each layer's weight and activation as well as performing multiply and accumulates (MAC) operations.

On the hardware side, Vivado synthesized the accelerator bitstream and analyzed the design architecture. Additionally, FINN-HLSLIB is one of the development tools utilized and built upon in this research. It is an open-source library to develop and implement an efficient QNN accelerator using AMD high-level synthesis (HLS). PYNQ is another open-source project-based tool used for Zynq platforms that provides a Python framework and APIs to load the bitstream and run the inference. Fig. 26 shows the steps followed in the training environment and hardware implementation.

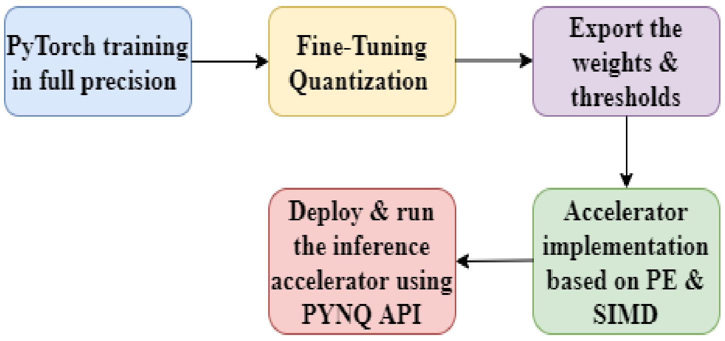
PyTorch training in full precision
Fine-Tuning Quantization
Export the weights & thresholds
Accelerator implementation based on PE & SIMD
Deploy & run the inference accelerator using PYNQ API

Figure 26. Block diagram of training and hardware environment steps.

After implementing and generating the results, the developed accelerator appeared to be very efficient due to the streaming dataflow architecture, on-chip weights storage, and being fully pipelined where each layer has its engine to speed up the processing, reduce the latency, and provide far more performance per watt than the GPU and CPU.

Implementation Results

Characteristics of hyperparameters during training play a significant role in achieving promising results of QDCGAN, in which trials and errors were the baselines of this research. MNIST and celebA datasets were trained from scratch on full precision, fine-tuned to the lowest possible bit-width, and achieved aesthetically clear generated images. Since MNIST is a greyscale dataset and has one channel, the lowest possible bit-width utilized were W1A2 and W4A4. Conversely, celebA is a large colored dataset with three channels, and the lowest possible bit-width utilized was W4A4. Moreover, the FID was employed to quantify the quality of generated images for various precisions. There is no specific range or scale for measuring the FID. However, a lower score indicates better-quality images. Hence, the generated images for both datasets are illustrated in Fig. 27 which shows different precisions with an associated FID score for each one.

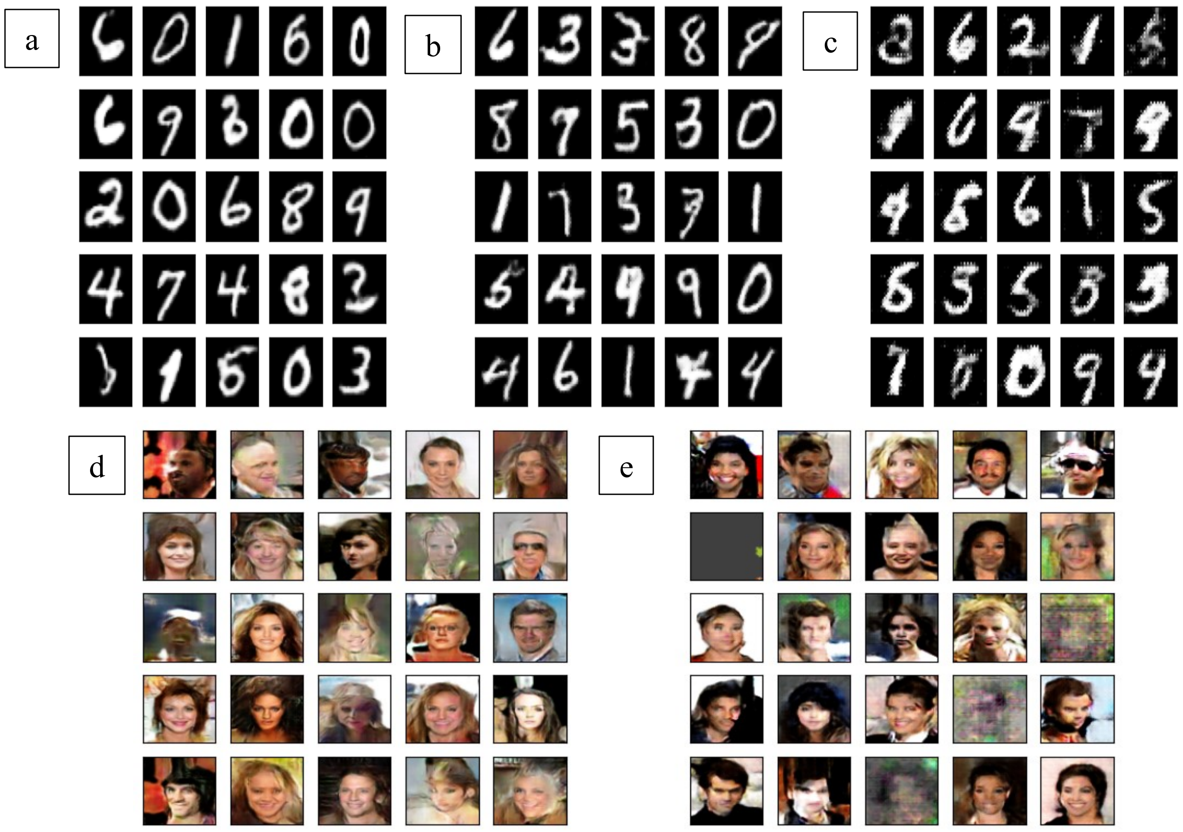

Figure 27. MNIST and celebA were trained from scratch on W32A32 for 100 epochs and resulted in (a) and (d) with an FID score of 49 and 104 respectively. Then, MNIST fine-tuned to a lower bit-width with more training epochs resulted in (b) W4A4 and (c) W1A2 with an FID score of 53 and 126 respectively. Finally, celebA is also fine-tuned to the lowest possible bit-width with more training epochs resulted in (e) W4A4 with an FID score of 129.

The developed accelerator-based deconvolution proved to be efficient in terms of less resource usage. Table 3 shows the resource utilization for Ultra96 and ZCU104 in runtime weights for both datasets. Moreover, running the inference with runtime configurable weights produces 2.1x-2.5x times higher throughput than the baked-in weights with less than 10% increase in the resource utilization reported. Lastly, the network architecture, scalability, throughput in frame per second (FPS), actual power consumption when running the inference, and the proposed accelerator efficiency (performance per watt) is reported and benchmarked against Jetson Nano in Table 4.

Table 3. Resource utilization.

Datasets	MNIST		celebA	
Device	Ultra96		Ultra96	ZCU104
Bit-width / resources	W1A2	W4A4	W4A4	
Flip-Flops	22k (15%)	141k (26%)	60k (42%)	63k (13%)
LUTs	17k (24%)	70k (46%)	44k (63%)	77k (33%)
BRAM	22 (10%)	48 (22%)	156 (72%)	174 (55%)
LUTRAM	1k (4%)	28k (5%)	1k (3%)	1k (1%)
DSP Slices	1 (0.28%)	1 (0.28%)	5 (1%)	5 (0.29 %)

Table 4. Performance measurement.

Dataset	MNIST		celebA		
Device	Jetson N	Ultra96	Jetson N	Ultra96	ZCU104
Output Channels	[128, 64, 32, 1]		[256, 128, 64, 32, 3]		
PE	-	[4,8,8,1]	-	[4,8,8,3]	[16,16,16,16,3]
SIMD	-	[4,16,16,8]	-	[4,16,16,16,8]	[16,16,16,16,16]
FPS	233-284	1802-1813	60-76	301-312	890-904
Power (W)	4.5-3.3 = 1.2	5.5-5.3 = 0.2	4.5-3.3 = 1.2	5.5-5.3 = 0.2	11.9-11.6 = 0.3
FPS/W	208	9K	58	1.5K	3k

Conclusion and Future Work

This chapter utilized AMD's publicly accessible FINN project to develop a scalable accelerator for QDCGANs. The proposed architecture is based on a ring buffer to generate the sliding windows for deconvolution resulted in higher throughput while using fewer resources. The provided open-source code can contribute to the community to explore and search efficient implementation of SRGAN on low-power FPGAs and assist in deploying NIVIDA DLSS on edge platforms which are considered as a solution for a wide range of medical and microscopic imaging applications. This work was published in the 2022 IEEE 56th Annual Conference on Information Sciences and Systems (CISS) [56].

CHAPTER FOUR: CONVOLUTIONAL NEURAL NETWORK ON THE EDGE

Introduction

CNNs have become the go-to architecture for various image-processing tasks, such as classification, object detection, and segmentation. CNNs are a class of deep neural networks that exploit the spatial structure of input data, typically images, by applying a series of convolutional, pooling, and fully connected layers. The core building block of a CNN is the convolutional layer, which consists of a set of learnable filters that slide across the input image, performing element-wise multiplications and summing the results to produce feature maps. These filters capture local patterns and features, such as edges, corners, and textures, which are essential for understanding the content of an image.

Convolutional layers are then followed by pooling layers, which reduce the spatial dimensions of the feature maps and introduce translation invariance. The pooled features are then passed through an activation function, such as ReLU, to introduce non-linearity. This process is repeated multiple times to capture high-level features. Lastly, the feature maps are flattened and passed through fully connected layers for high-level reasoning and producing the output, such as class probabilities for classification tasks. This combination of layers shown in Fig 28 allows CNNs to learn hierarchical representations and achieve state-of-the-art performance on computer vision tasks.

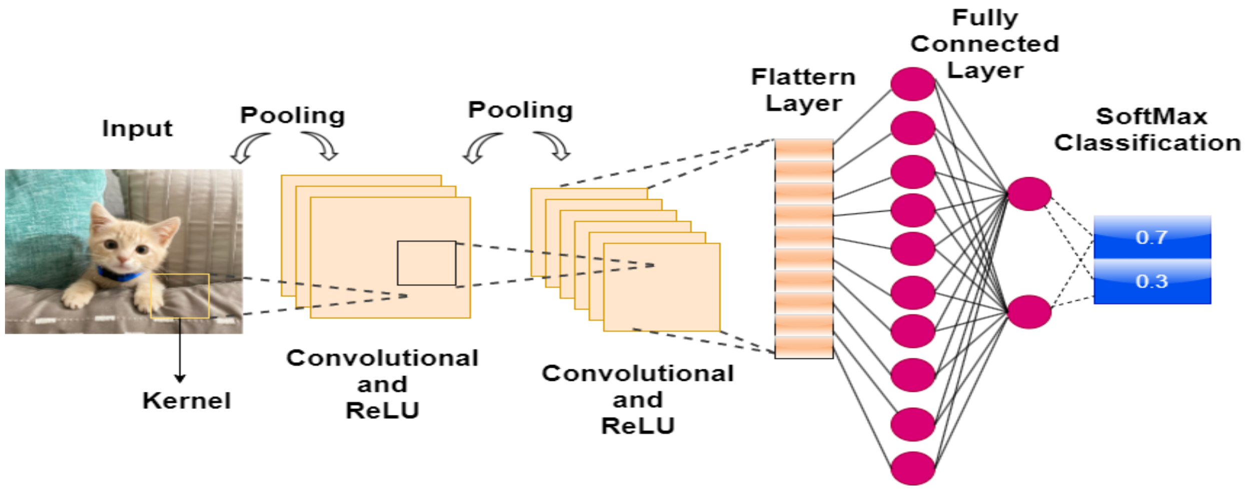

Figure 28. Traditional CNN architecture.

Additionally, the success of CNN algorithms in vision tasks heavily relies on effectively training of the network parameters. Various training techniques have been proposed to improve the convergence, generalization, and robustness of CNN applications. The most common techniques are:

- Data augmentation involves applying random transformations to the input images, such as rotation, scaling, cropping, and flipping [76].
- Transfer learning leverages pre-trained models on large-scale datasets, such as ImageNet, to initialize the weights of a new CNN [93].
- Regularization and dropout are also crucial for preventing overfitting and improving the generalization of CNNs [218].
- Learning rate scheduling improve the convergence and stability of CNN training.
- Batch normalization is widely used in each layer to have zero mean and unit variance, which help reduces the internal covariate shift and allows for higher learning rates [82].

These adaptive optimization algorithms can accelerate the convergence and improve the stability of the training process. As a result, CNNs have achieved remarkable accuracy on vision tasks. One of the most notable achievements is the performance of CNNs on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which involves classifying images into 1000 object categories [57]. In 2012, the AlexNet architecture achieved a top-5 error rate of 15.3% on the ILSVRC, significantly outperforming the previous methods [219]. This marked a turning point in computer vision and sparked the widespread adoption of CNNs. Since then, numerous CNN architectures have been proposed, each pushing the boundaries of accuracy on the ImageNet benchmark. Table 5 is shows that some SOTA architectures achieved top-1 accuracy on ImageNet alongside other benchmarking and performance metrics.

Table 5. CNN architectures performance results on ImageNet measured on V100 GPU [4].

Model	Top-1 Acc	Params	FLOPs	Inference time (ms)
EfficientNet-B3	81.5%	12M	1.9M	19
EfficientNet-B7	84.7%	66M	38B	170
RegNetY-8GF	81.7	39M	8B	21
RegNetY-16GF	82.9%	84M	16B	32
ResNet-101	83.0%	48M	13B	31
ResNet-200	83.9%	70M	36B	76
EfficientNet-X	84.7%	73M	91B	-
NFNet-F0	83.6%	72M	12B	30
NFNet-F4	85.9%	316M	215B	309
EfficientNetV2-S	83.9%	22M	8.8B	24
EfficientNetV2-L	85.7%	120M	53B	98

CNN Challenges in Edge Computing

Despite their remarkable performance, CNNs face significant computational challenges, mainly when deployed on resource-constrained edge devices. The primary challenge stems from the many of parameters and the intensive matrix multiplications required in convolutional layers. As CNNs become more profound and complex, the computational demands increase exponentially, making real-time inference on edge devices daunting. Various techniques have been proposed to address these challenges, including network pruning, quantization, and knowledge distillation [6]. These methods reduce the model size and computational complexity while maintaining acceptable accuracy. Another approach to mitigate the computational burden of CNNs is to design efficient architectures that balance accuracy and complexity. Architectures such as MobileNet [142], ShuffleNet [145], and EfficientNet [7] have been proposed to enable real-time inference on mobile and embedded devices by leveraging depthwise separable convolutions, channel shuffling, and neural architecture search.

Hardware acceleration using specialized architectures such as GPUs, FPGAs, and ASICs has also been explored to speed up CNN inference [220]. GPUs consist of many programmable cores that efficiently execute the matrix multiplications and convolutions required in CNNs [221]. The availability of high-level programming frameworks, such as CUDA and OpenCL, has made GPUs accessible to a broad range of developers. However, GPUs are known to consume significant power, which can be a limitation for energy-constrained systems. Recent advancements in GPU architectures, such as NVIDIA's Tensor Cores and AMD's Radeon Instinct, have focused on improving CNN inference performance and energy efficiency but are still considered power-hungry for edge devices.

ASICs, on the other hand, are custom-designed circuits that are explicitly tailored for CNN inference [40]. ASICs offer the highest performance and energy efficiency among the three platforms, as they can be optimized at the transistor level for the specific computation patterns of

CNNs. Examples of ASIC-based CNN accelerators include Google's TPU [222] and Intel's Nervana Neural Network Processor (NNP) [134]. ASICs provide superior performance and energy efficiency compared to GPUs and FPGAs, but they lack flexibility and require a high development cost and time. Moreover, ASICs are fixed-function devices that cannot be easily updated to support new CNN models or algorithms.

FPGAs balance the flexibility of GPUs and the efficiency of ASICs [66]. FPGAs consist of an extensive programmable logic blocks and interconnects that can be reconfigured to implement custom hardware accelerators. This reconfigurability allows FPGAs to be adapted to different CNN models and optimized for specific performance and energy requirements. FPGAs have lower power consumption compared to GPUs and can customized to achieve higher performance than ASICs for specific CNN workloads.

Related work

Several studies have conducted benchmarking and comparison of these hardware platforms for CNN inference. Nurvitadhi et al. [223] compared the performance and energy efficiency of GPU, ASIC, and FPGA platforms for a range of CNN models. They found that ASICs achieved the highest performance and energy efficiency, followed by FPGAs and GPUs. However, the performance gap between FPGAs and ASICs narrowed for larger CNN models, indicating the potential of FPGAs for scalable CNN inference.

Jouppi et al. [27] presented a detailed analysis of Google's TPU ASIC and compared its performance and energy efficiency to contemporary GPUs and CPUs. They demonstrated that the TPU achieved 15x - 30x higher performance per watt than GPUs and CPUs for CNN inference workloads. The TPU's architecture was optimized explicitly for the dataflow and memory access patterns of CNNs, enabling significant gains in performance and energy efficiency.

Qasaimeh et al. [224] surveyed and compared of FPGA-based CNN accelerators. They analyzed various FPGA architectures, design methodologies, and optimization techniques for CNN inference. The study highlighted the diversity of FPGA-based CNN accelerator designs and their trade-offs in performance, energy efficiency, resource utilization, and flexibility. The authors emphasized the need for standardized benchmarking and evaluation methodologies to facilitate fair comparisons among different FPGA-based CNN accelerators.

Objective

Previous researchers have reported significant issues on the scalability side. First, different CNN architectures have different layers' parameters that complicate the accelerator design. Second, some platforms need more scalability due to their restricted resources. However, the most recent work in line with our idea is proposed by Bjerge et al. [179], which implemented a 16-bit quantized CNN in 2.14 format of two-bits integer and fourteen-bits fractional using a PYNQ framework. Nevertheless, this paper is qualitatively distinct from the past ones. Our proposed methodology addressed these issues, optimized performance, latency, and resource utilization, and benchmarked against previous developments.

This research aims to find common patterns among two algorithms, create an HW/SW partitioning scheme, and then develop an efficient and scalable accelerator on the PL to compute the intensive operations of convolution and FC layers and gain higher performance. On the contrary, network initialization, pooling, normalization, SoftMax, and other layers are performed on the PS. The contributions of this study are as follows:

- Computed convolutional and FC layers operations in vector multiplication on a single on-chip compute unit for AlexNet, VGG16, and LeNet architectures.
- We Utilized loop tiling transformation efficiently to construct the IP accelerator core.
- We have demonstrated that the proposed methodology achieved superior performance up to 230 GOP/s under 200-MHz with minimum data execution time.

Methodology

Network Architecture

CNN architecture, as shown in Fig. 29, is used as a case study for the proposed template.

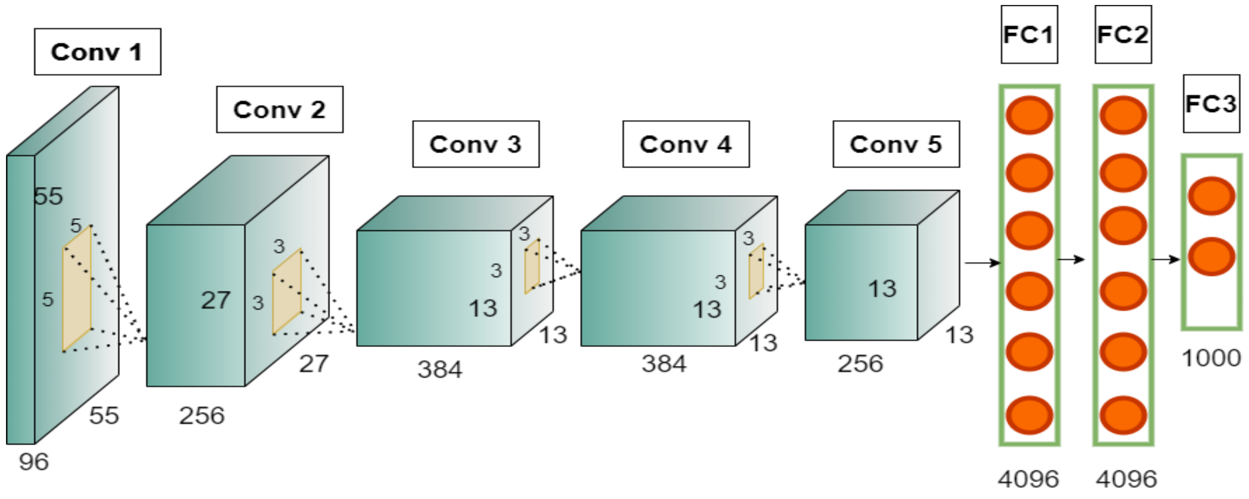

Figure 29. CNN AlexNet architecture.

All these networks consist of convolutional and FC layers, the number and size of these layers vary across different operations. The algorithm of convolutional layers is illustrated in the equation follows:

$$\forall\,\text{row} \in \{1, 2, \ldots, \text{R}\}$$
$$\forall\text{col} \in \{1, 2, \ldots, \text{C}\}$$
$$\forall\text{co} \in \{1, 2, \ldots, \mathcal{q}\}$$
$$\forall\text{ci} \in \{1, 2, \ldots, \mathcal{p}\}$$
$$\forall\text{i} \in \{3,5, \ldots, \mathcal{K}\}$$
$$\forall\text{j} \in \{3,5, \ldots, \mathcal{K}\}$$
$$OFM\,[row{:}\,col{:}\,co] = \sum\sum\sum \begin{matrix} IFM[s*row\ +\ i][s\ *\ col \\ +\,j][ci]\;W\;[co][ci][i][j] \end{matrix} \tag{4.1}$$

Rows (R) and columns (C) represent the image/matrix size. At the same time, the input channel *($\mathcal{p}$)* and output channel *($\mathcal{q}$)* are the third dimensions of input feature map (IFM) and output feature map (OFM), respectively, and *($\mathcal{K}$)* represents the kernel/filter size. The total number of convolutional operations is as in the equation follows:

$$No.of\ CONV\ Operations\ = \prod 2RC\mathcal{p}\mathcal{q}\,\mathcal{K}^2 \tag{4.2}$$

On the other hand, the algorithm of FC layers is illustrated in the equation follows:

$$\forall\text{co} \in \{1, 2, \ldots, \mathcal{q}\}$$
$$\forall\text{ci} \in \{1, 2, \ldots, \mathcal{p}\}$$

$$OFM\,[\text{co}] = \sum\sum IFM[\text{ci}] \times \text{W}[\text{co}][\text{ci}] \tag{4.3}$$

Where the operations of FC Layers have one-dimensional data on the input and output neurons, which have fewer processing tasks compared to the convolutional layers. The total number of FC operations is as in the equation as follows:

$$No.of\ FC\ Operations = \prod 2\mathcal{p}\mathcal{q} \tag{4.4}$$

As previously mentioned, these layers are operationally expensive regarding computation and latency. For this reason, it is pertinent to map them into the PL part of ZYNQ. The previous equations represent symmetrical operation, while the dynamics and dataflow of the layers are different. The proposed methodology finds a common dataflow pattern for an optimized accelerator design.

Loop Tiling Transformation

Equation 4.1 represents layer values of $R, C, \mathscr{p}, \mathscr{q}$ *and* $\mathcal{K}$ as variable values in which using these values to perform direct implantation leads to an inefficient accelerator design. Hence, loop tiling is performed, which converts loops into fixed points/blocks. It is also represented in the same algorithm and uses the tile size as $(\mathcal{T})$ for R, $(\mathfrak{C})$ for C, (μ) for $\mathscr{p}$, and (τ) for $\mathscr{q}$. Thus, we can transfer a fixed amount of data from the external memory (DRAM) to on-chip memory (BRAM). Once the data is cashed into BRAM, fixed computations are performed by the accelerator. On the contrary, loop tiling for the FC layers uses the tile size as (λ) for $\mathscr{p}$ and (Ω) for $\mathscr{q}$.

In the accelerator design, the FC layers have larger vector values in the input and output channels compared to the convolutional layers. Hence, different sizes of tiles are chosen. However, choosing the same size of tiles for the input and output channels resulted in performance reduction. Lastly, the overall selection of these tiles can result in maximum resource utilization and lower latency.

Accelerator Design

The proposed template-based vector design is illustrated in Fig. 30, which shows the loop tiling factors of $\mathcal{T}, \mathfrak{C}, \mu, \tau, \lambda$, and Ω that determine the on-chip buffers. Due to the size difference of input and output channels in convolutional and FC layers, different sizes of tiles are used and prompted to use dedicated buffers for both types of layers. This method resulted in more resource utilization and overcome the reading and writing overhead latency owing to the multi-dimensional array. Additionally, those dedicated buffers are being used in the size of weights in both layers which allow the design to have better efficiency and lower latency at the cost of more resource utilization.

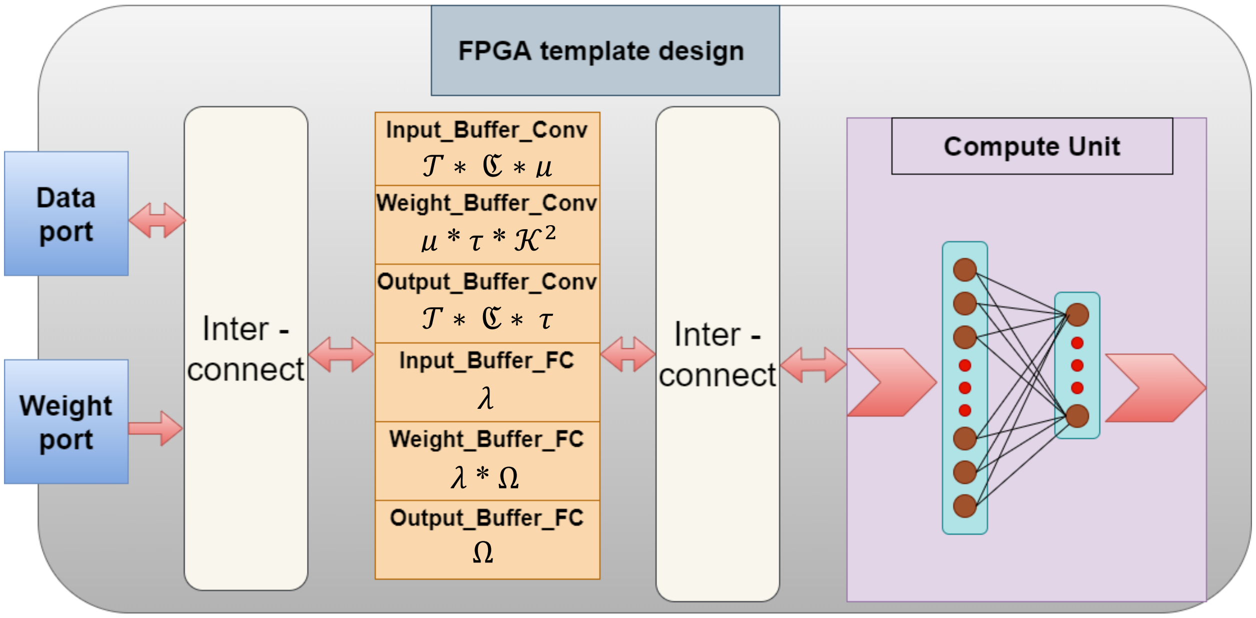

Figure 30. The proposed FPGA template-based design.

First, data is cashed on-chip in these buffers using two data ports where one can be used for read/write of (IFM) and (OFM), and the other one can be used only for reading the weights. Those ports are memory-mapped (M-AXI), which enables a burst transfers and improv the external memory bandwidth. The input and weight buffer are partitioned in dimension τ to improve the design latency as multiple reads and writes are possible on arrays. This proposed compute unit is designed to do a dot-product between μ and τ input/output neurons respectively.

The template design has a scheduling mechanism among the interconnect, which orchestrate control logic and dataflow. Furthermore, the ping pong data transfer method is used on the input, weight, and output buffers to ensure a simultaneous data transfer happens from DRAM to on-chip buffers and then from on-chip buffers to the compute unit. Lastly, equations of the total operations performed by IP accelerator in Eq. 4.5 and performance measurement in Giga Operations per second (GOP/s) in Eq. 4.6 are as follows:

$$Accelerator\ IP = \mathcal{I}_{\mathcal{P}} = \prod 2\, \mathcal{T}\, \mathfrak{C}\, \mu\, \tau\, \mathcal{K}^2 \tag{4.5}$$

$$Performance\ GOP/s = \mathcal{G}_{\mathcal{PS}} = \frac{\mathcal{I}_{\mathcal{P}}}{Latency} \tag{4.6}$$

Dataflow Modeling

Dataflow modeling is done concerning the architectural details of on-chip buffers and the compute unit. Convolutional algorithm in Eq. 4.1 is based on window operation in which $\mathcal{K} * \mathcal{K}$ weight window is convolved with $\mathcal{K} * \mathcal{K}$ patch of the input pixel of the (IFM). The sum of these operations is resulted in the (OFM) at a particular index. This straightforward approach has a complex data pattern on the FPGA which produce poor architecture design. On the contrary, the FPGA can parallelize the workload, so layers dataflow is simplified and present low dependency from the on-chip buffers to the compute unit.

The dataflow modeling of convolutional layers in Fig. 31 is shown $\mathcal{T} * \mathfrak{C} * \mu$ as an input neuron, $\mathcal{T} * \mathfrak{C} * \tau$ as an output neuron, and $\mu * \tau * \mathcal{K}^2$ as a weight value where all of them are cashed into input, output, and weight buffers for on-chip processing. First, the data is moved from the input and weight buffers to the compute unit to perform dot product which and resulted in a written values in the output buffer. Then, the dataflow occurs in a form of vector values across those channels. All (IFM) values across channels (0, 1, 2…, μ-1) are read starting from the index (0,0) to the last index ($\mathcal{T}$-1, $\mathfrak{C}$-1) and then transferred alongside with the weight's values to the compute unit. After that, the compute unit performs dot product along the channel dimension of (OFM) and resulted in the output vector (0, 1, 2…, τ-1). This process is continuously repeated for a spatial location of $\mathcal{K} * \mathcal{K}$ on (IFM) and then stored on (OFM) to achieve high parallelism in the dimension of input and output channels (μ, τ) and reduce data dependency for reading and writing among those buffers.

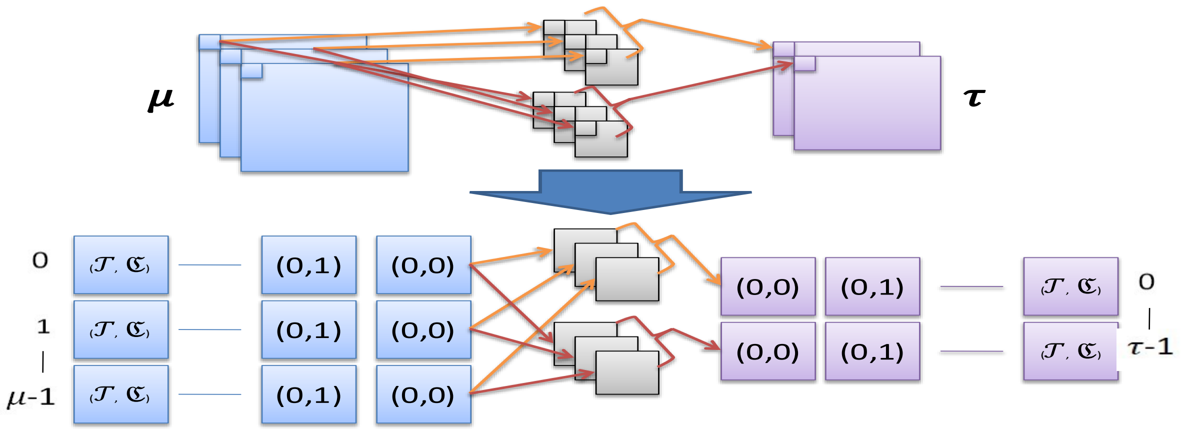

Figure 31. Depiction of Convolution dataflow and computation on FPGA.

On the other hand, FC layers working principle is illustrated in Fig. 32 shown λ input neurons and $\lambda * \Omega$ weight values are cashed on the BRAM buffers. These values are too large to be processed by the compute unit at once. As a result, another set of loop tiling/block is introduced for the FC layers which break (λ, Ω) data into smaller (μ, τ) sizes. The input size μ and the weights size $\mu * \tau$ are transferred to the compute unit while the output size τ is written back to the output buffer. This method ensure that the entire input vector is processed efficiently on the same compute unit.

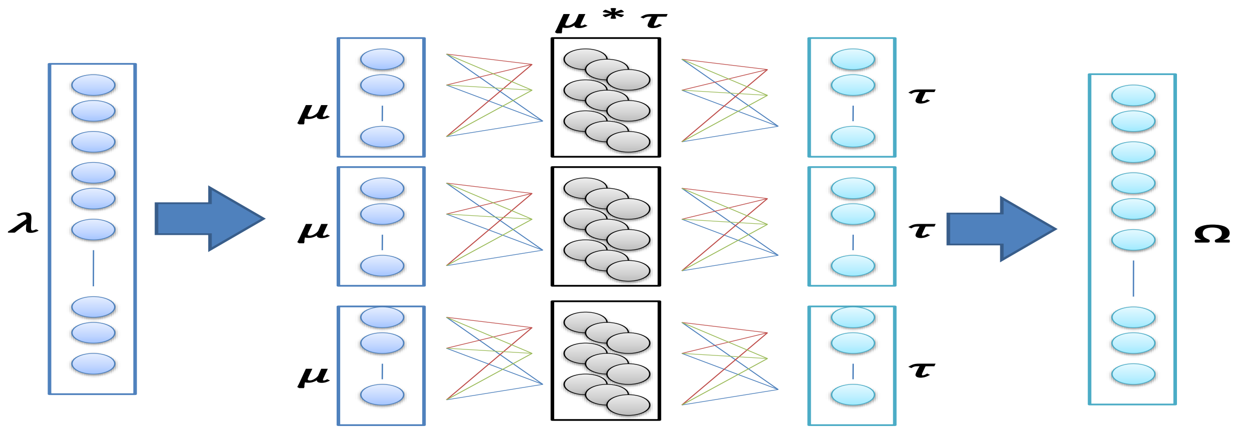

Figure 32. Depiction of Fully Connected dataflow and computation on FPGA.

Scalability and Efficiency

The proposed accelerator utilized a pre-trained model with 16-bit fixed-point quantization in 2.14 format. Nevertheless, the tiling size of convolution and FC layers determines the optimum performance of the template. We used a trial-based method to fine-tune the accelerator for higher efficiency and better scalability across various SoC-FPGAs ZYNQ boards and CNNs architecture. The parameters were randomly selected, and the design was simulated until the resources and latency were met. After sets of trial and error, we found that the tile factor of μ*τ affect the performance directly in which this proposed template achieved higher performance when τ is approximately twice μ under resource constraints. As a result, the accelerator can run advanced CNN architecture such as ResNet-50, SSD, MobileNets and YOLO of any version. Since all of these networks have same type of layers, the proposed methodology can map any CNN architecture.

Experimental Setup and Results

Development Environment

The accelerator design was simulated and synthesized using Vivado HLS (2019.2), and utilized pre-trained models from PyTorch Model Zoo. It was tested with AlexNet, VGG-16, and LeNet architectures and can work with any advanced CNNs network.

Results

The proposed template operated under 200MHz, and achieved superior performance of up to 230 GOP/s. Table 6 reported the resource utilization and performance measurement of AlexNet network demonstrated on Ultra96, ZCU104, and ZCU102. The BRAM and DSP were directly dependent on the tile size of $\mathcal{T}$ and $\mathfrak{C}$, and the number of dot-products in the compute unit, while FF and LUT were used to control the logic gates and state machine for running the loops and controlling the dataflow. Finally, our accelerator was benchmarked against the previous development [179] on Ultra96 for performance and lower latency, as reported in Table 7.

Table 6. Resource utilization and performance measuremment for AlexNet architecture.

Device	**Ultra96**	**ZCU104**	**ZCU102**
Compute Unit $\mu * \tau$	12 x 24	20 x 30	20 x 55
Flip-Flops	23.5k (16%)	46k (10%)	139k (25%)
LUTs	15.6k (22%)	24k (10%)	57k (20%)
BRAM	332 (76%)	594 (95%)	1.7K (95%)
DSP Slices	334 (92%)	586 (33%)	1.7K (67%)
Performance	51 GOP/s	107 GOP/s	230 GOP/s
Frequency	169 MHz	198 MHz	167 MHz

Table 7. Benchmarking and comparision.

Device	Ultra96	
Development	Previous method [179]	Proposed method
Max frequency	170 MHz	169 MHz
Bit width	16	16
Performance	31 GOP/s	51 GOP/s
Latency (ms)	4.6	0.174
Power (w)	3.55	4.7

Conclusion

The proposed template efficiently utilized the loop tiling and dataflow modeling for optimized accelerator design. As a result, a range of 1.3x - 1.7x higher performance was achieved along with a minimal layer of execution time when compared to the previous development. The analysis and simulation results proved to be optimistic and can be extended to create a complete framework. This will allow the community to use our open-source project and search an efficient implementation for real-time applications [59]. This work was published in the 2022 IEEE International Conference on Networking, Architecture and Storage (NAS) [225].

CHAPTER FIVE: HUMAN ACTION RECOGNITION ON THE EDGE

Introduction

Human Action Recognition (HAR) is a crucial task in computer vision that aims to understand human interaction in video representation and time-series data automatically. In recent years, different DNN algorithms have achieved SOTA performance on various HAR datasets. Three prominent DNN architectures for HAR are: 3D-CNNs, Graph Convolutional Networks (GCNs), and two-stream CNNs. Fig. 33 is shows a simple DNN network and how HAR is constructed.

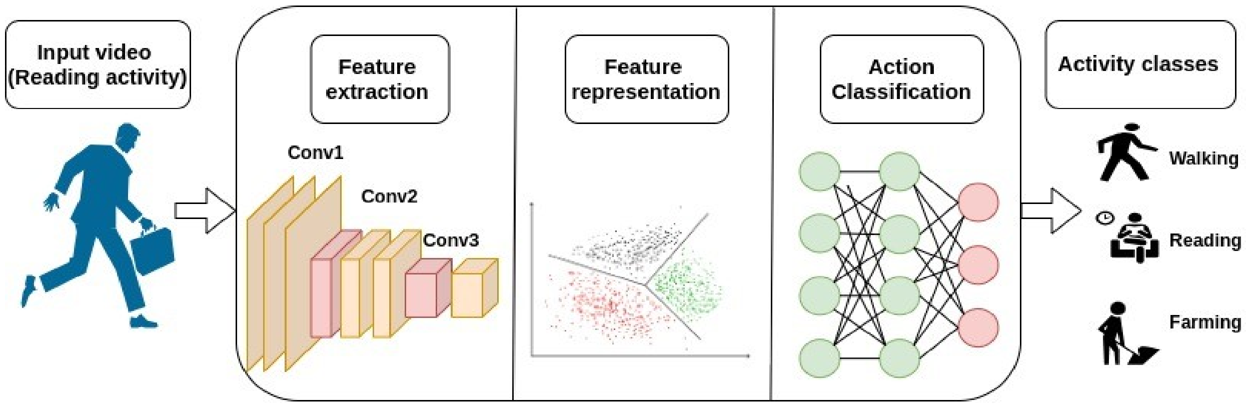

Figure 33. Human Action Recognition CNN-based framework architecture.

First, 3D-CNNs extend the traditional 2D convolution operation to the temporal dimension, enabling them to learn spatio-temporal features directly from video data [226]. 3D-CNNs can effectively capture the motion and appearance cues necessary for action recognition by considering spatial and temporal information. One of the seminal works in this area is the C3D network [227], which employs 3D convolutions with a fixed kernel size of $3x3x3$ and demonstrates impressive performance on various HAR benchmarks. To train 3D-CNNs effectively, large-scale datasets such as Kinetics and UCF101 are commonly used. Data augmentation techniques, including random cropping, flipping, and temporal jittering, are applied to enhance the diversity of the training data and improve generalization.

Recent advancements in 3D-CNNs include the Inflated 3D ConvNet (I3D) [228], which leverages pre-trained 2D CNN weights and inflates them into 3D kernels, allowing for efficient training and improved accuracy. Another notable architecture is the R(2+1)D network [229], which factorizes the 3D convolution into separate spatial and temporal convolutions, reducing computational complexity while maintaining performance.

Similarly, GCNs have emerged as powerful tool for modeling structured data, such as human skeleton sequences in HAR tasks. By representing human joints as nodes in a graph and their connections as edges, GCNs can effectively capture the spatial and temporal dependencies between body parts. Attention mechanisms have been incorporated into the training of GCNs to focus on the most informative joints and time steps.

The seminal work by Yan et al. [230] introduced the Spatial-Temporal (ST-GCN), which applies graph convolutions on skeleton sequences to learn both spatial and temporal patterns for action recognition. Subsequent works have extended the ST-GCN architecture to improve its representational power and robustness. For example, the Two-Stream Adaptive (2S-AGCN) [231] introduced an adaptive layer to capture richer dependencies between joints and a two-stream framework to fuse both skeleton and pose information for higher accuracy on skeleton-based HAR datasets.

On the other hand, two-stream CNNs introduced by Simonyan and Zisserman [232] have become a popular approach for HAR by processing both spatial (appearance) and temporal (motion) information separately. The spatial stream operates on individual frames to capture static appearance cues, while the temporal stream operates on optical flow fields to capture motion information. RGB frames and optical flow are utilized to train two-stream CNNs. The spatial stream is typically pre-trained on large-scale image datasets like ImageNet, while the optical flow is computed using algorithms like TV-L1 [233].

The outputs of both streams are then fused to make the final prediction. Variants of the two-stream architecture have been proposed to improve its performance and efficiency. The Temporal Segment Network (TSN) [234] extends the two-stream framework by sparsely sampling frames from different video segments, enabling the network to capture a long-range temporal structure that helped robust performance on challenging video datasets.

Deep learning-based methods such as CNNs and GNN have shown promising results in capturing the spatial and temporal dependencies of human actions. Transfer learning and domain adaptation techniques have been used to leverage knowledge from related domains and adapt models to new environments. Attention mechanisms and graph-based representations have been explored to focus on relevant regions and model the relationships between body parts.

However, many datasets used for performance measurement to evaluate HAR models, and confusion matrix and recognition accuracy are the commonly used criteria. The confusion matrix indicates the detailed recognition results between each category, while the recognition accuracy is the ratio of the number of correctly recognized data elements to the total number of test data elements. Table 8 presents the benchmarking accuracy among different datasets.

Table 8. Recognition accuracies of methods on RGB datasets [235].

Methods	Hollywwod2	HMDB51	Olympic Sports	UCF101	Kinetic
Motion Vectors	-	-	-	86.4%	-
ST-GCN	-	-	-	-	81.5%
3D-Scale	68.1%	-	94%	-	-
Hidden-Two Stream	-	78.7%	-	97.1%	-
3D-ConvNets	-	63.5%	-	93.2%	-
Deep Local	-	75%	-	95.3%	-
Kinetics HAR	-	-	-	-	79%
Two Stream- DTPP	-	74.8%	-	95.8%	-
I3D models	80.2%	-	-	97.9%	-

HAR Challenges in Edge Computing

Implementing HAR on edge devices poses several challenges due to their limited computational resources, memory constraints, and power consumption requirements. Traditional HAR approaches, such as 3D-CNNs GCN, require substantial computational resources and memory, making them unsuitable for deployment on resource-constrained edge devices. Moreover, the limited storage capacity of edge devices restricts the size of the DNN models. In addition, edge computing scenarios often demand low-latency and real-time responses, especially in applications like video surveillance and human-robot interaction, which makes it difficult to achieve real-time performance while maintaining high accuracy [236].

Two-stream CNNs have emerged as an efficient and effective solution for HAR on edge devices to address these challenges. It consists of two separate networks: a spatial stream that processes individual frames to capture appearance information and a temporal stream that operates on optical flow to capture motion information. By decoupling the spatial and temporal processing, two-stream CNNs can balance accuracy and computational efficiency well. The spatial stream of a two-stream CNN can be implemented using lightweight CNN architectures, such as MobileNet [142] or ShuffleNet [145], specifically designed for edge devices. These architectures employ techniques like depthwise separable convolutions and channel shuffling to reduce the computational complexity and model size while maintaining high accuracy. On the other hand, the temporal stream can be realized using efficient optical flow estimation methods, such as FlowNet [237] or LiteFlowNet [238], which can be optimized for real-time performance on edge devices. By employing previous techniques, a two-stream CNN's the spatial and temporal streams can be pre-trained on large-scale action recognition datasets such as Kinetics [239] or UCF101 [240]. These pre-trained models can then be fine-tuned on the target HAR dataset, requiring less training data and computational resources than training from scratch. Transfer learning not only improves the accuracy of the models but also reduces the training time and computational requirements on edge devices.

Related Work

Numerous studies have investigated the implementation of HAR on edge devices using various approaches, and several benchmarking studies have been conducted to evaluate the performance of two-stream CNNs on different edge hardware platforms. Nooruddin et al. [241] benchmarked the inference speed and accuracy of two-stream CNNs on various edge devices, including Raspberry Pi, NVIDIA Jetson Nano, and Intel NCS. They found that the NVIDIA Jetson

Nano achieved the highest inference speed, processing up to 40 frames per second (FPS) for real-time HAR, while the Intel NCS provided the best balance between speed and power consumption.

Sarabu et al. [242] conducted a comprehensive benchmarking study of two-stream CNNs on the AMD ZCU102 FPGA board. They implemented the spatial and temporal stream CNNs using HLS and optimized the design for performance and resource utilization. Their FPGA implementation achieved an inference speed of 60 FPS, outperforming the GPU implementation in terms of both speed and energy efficiency.

Sun et al. [243] benchmarked the performance of two-stream CNNs on the Google Coral Edge TPU, a purpose-built ASIC for edge AI applications. They quantized the models to 8-bit precision and deployed them on the Edge TPU using the TensorFlow Lite framework. The Edge TPU implementation achieved an impressive inference speed of 200 FPS, enabling real-time HAR with high accuracy and low latency.

Objective

Several embedded platforms, such as ASIC, NVIDIA Jetson Nano, Raspberry Pi, and AMD Kria KV260, have been utilized to accelerate the two-stream CNN on the edge. Among those, the AMD SoC-FPGA stands out as the only hardware that offers PL fabric coupled with an ARM-based PS. This unique feature allows developers to reconfigure the PL fabric part and synthesize deeply pipelined custom accelerators for any algorithm. The SoC-FPGAs also allow exploring different precision optimizations to trade-off between latency, throughput, and power.

Moreover, the most researchers and open-source efforts focus on utilizing 3D-CNN and GCN algorithms in order to achieve extraordinary HAR accuracy on large FPGA boards. However, these algorithms have a higher number of parameters and require more compute resources, which is inefficient for MAC operation and on-chip (BRAM) memory units on edge FPGAs. For this reason, the developed accelerator for two-stream CNN-based achieved a balanced accuracy and optimal performance for real-time inference on edge FPGAs. As a result, an efficient implementation of HAR opens the possibility of having technologies like Tesla-Autopilot deployed on low-power computing platforms.

The most recent work that is in line with this study is proposed by Lin et al. [244], which implemented an 8-bit quantized two-stream VGG7-CNN with ResNet-18 backbone on a large FPGA board (ZCU102) and achieved 12-15 FPS. Nevertheless, this research is constrained to; First, it showed one dataset result without experimenting with the network effect on smaller datasets. Second, the accelerator is not fully optimized for real-time performance, which implies that some layers are executed sequentially on the PS, causing in increased latency. Third, the model weights require larger boards and more available resources (ex. ZCU102), resulting in more power consumption.

In this research project, we adopted the SimpleNet-PyTorch architecture [245] and extended it with QAT two-stream CNN. We also designed a scalable accelerator, deployed the inference on the edge, and benchmarked our design against several hardware platforms. Our key contributions are as follows:

- Parallelized the two-stream HAR SimpleNet workload by incorporating an improved fusion layer [246], consolidating all convolutional layers with batch-norm and ReLU into a single homogeneous five-layers structure, and utilizing Lucas-Kanade optical flow algorithm in training and implementation [247] on UCF101 and UCF24 datasets.

- Developed an open-source scalable and customizable inference accelerator known as (FPGA-QHAR) [248] and deployed it with PYNQ image on the ZCU104 board.
- Demonstrated that the proposed methodology achieved nearly 81% accuracy and real-time throughput up to 24 FPS under 187MHz with a performance up to 120 GOP/s.

Lastly, our proposed methodology addressed these issues by balancing the network accuracy and accelerator performance to meet the needs for smaller edge devices in real-time applications.

Methodology

Network Architecture

To achieve optimal efficiency in both resource usage and algorithm acceleration, our customized architecture (SimpleNet) shown in Fig. 34 employs two-stream lightweight CNNs with a design process of homogeneously stacking several types of layers such as Convolutional, Batch-Normalization, and ReLU in one group layer. This approach [249] allows us to easily manage the number of parameters in the network while providing better max pooling information for each semantic level. Moreover, the homogeneous layer group makes the network compression via layer fusion possible [250] to meet the constraint of edge FPGAs resources.

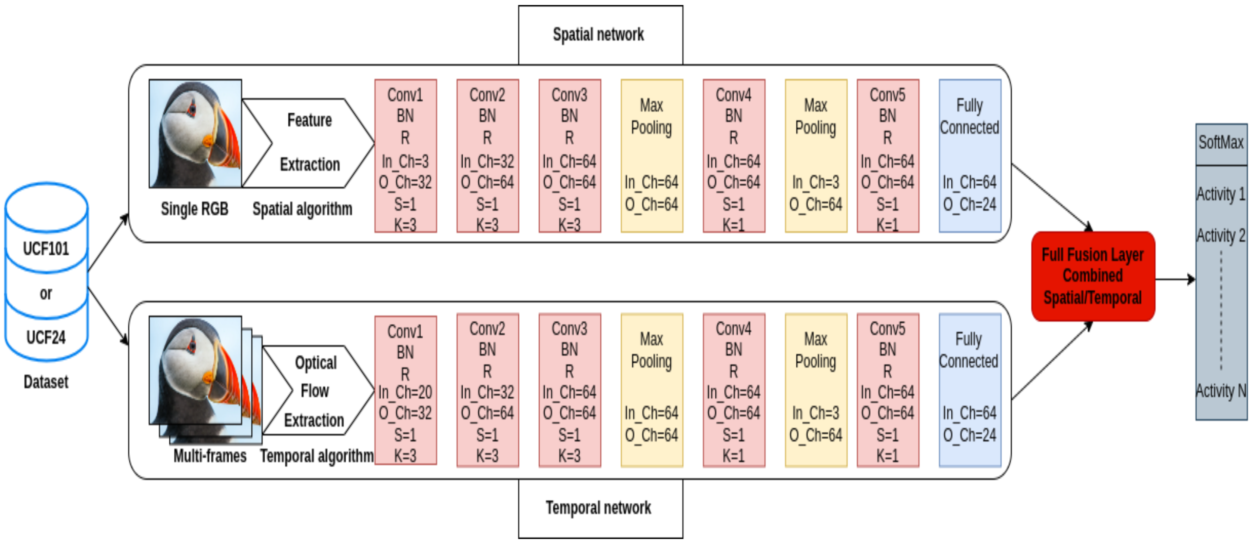
Spatial network
Temporal network
UCF101
or
UCF24
Dataset
Single RGB
Feature Extraction
Spatial algorithm
Multi-frames
Optical Flow Extraction
Temporal algorithm
Conv1 BN R In_Ch=3 O_Ch=32 S=1 K=3
Conv2 BN R In_Ch=32 O_Ch=64 S=1 K=3
Conv3 BN R In_Ch=64 O_Ch=64 S=1 K=3
Max Pooling In_Ch=64 O_Ch=64
Conv4 BN R In_Ch=64 O_Ch=64 S=1 K=1
Max Pooling In_Ch=3 O_Ch=64
Conv5 BN R In_Ch=64 O_Ch=64 S=1 K=1
Fully Connected In_Ch=64 O_Ch=24
Conv1 BN R In_Ch=20 O_Ch=32 S=1 K=3
Full Fusion Layer Combined Spatial/Temporal
SoftMax
Activity 1
Activity 2
Activity N

Figure 34. The proposed QAT two-stream CNN architecture combining Convolutional (Conv), Batch-Normalization (BN), and ReUL into single five layers. Input-channel (In_Ch), Output-Channel (O_Ch), Stride (S), and Kernel (K) determined the size of each layer.

The CNN computations of IFM and OFM represent the input feature maps and the output feature maps, respectively, and can be mathematically expressed as shown in the equation follows:

$$OFM = \sum_{n=1}^{N} \sum_{k=1}^{k*k} IFM[n][k] \times W[n][k] + Bias \qquad (5.1)$$

Where N represents the number of IFM channels, K is the kernel size, and W is matrix weights. This operation is followed by BN to improve the training speed, accelerate the convergence, and reduce the insensitivity initialization weights of the network. It can be expressed as follows:

$$x = \gamma \frac{OFM - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta \qquad (5.2)$$

Where μ and σ^2 represent the mean and variance, respectively, x denotes the output pixel after BN, γ refers to the scale coefficient; β is the offset coefficient, and ε indicates a very small positive number. The results are then passed through ReLU to introduces non-linearity into the network and learn complex relationships in the data, as in the equation follows:

$$y = \begin{cases} x, & x \geq 0 \\ 0.1x, & x < 0 \end{cases} \quad , \quad y = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{5.3}$$

Adding a pooling layer in the network after the homogeneous group helps to reduce overfitting by indicating the maximum value of the selected region. After that, the OFM of pooling is the IFM of the next homogeneous layer, which is expressed as in the following equation:

$$f = \{\max\,(y_1, y_2,, y_3, \dots, y_n),\ \&\max pooling \tag{5.4}$$

Those operations are then followed by FC layers to perform nonlinear transformations on the features extracted by convolutional layers. In addition, the Full Fusion linear layer is introduced to minimize the computational cost of the network by combining the neurons of the spatial and temporal networks, while also enabling the learning of more complex data relationships. Subsequently, the SoftMax layer generates a probability distribution (value number Z) over the possible classes as indicated in the equation follows:

$$softmax(z_i) = \frac{exp(z_i)}{\sum_i exp(z_i)} \tag{5.5}$$

Finally, the cross-entropy loss function shown in Eq. 5.6, is used to measure the dissimilarity between the predicted probabilities and the true labels, while the Stochastic Gradient Descent (SGD) shown in Eq. 5.7, is computed during backpropagation to minimize the losses by updating the parameters of the network.

$$\mathcal{L}(\theta) = -\sum_{i=0}^{n} \hat{y}_i \,.\log(y_i) \ \Big| \begin{array}{l} y_i{:}\, prediction\ vector\ of\ \vec{y} \\ \hat{y}_i{:}\ ground\ truth\ label\ of\ \hat{y} \end{array} \tag{5.6}$$

$$\theta = \theta - \alpha \nabla_{\theta} J(\theta; x^{(i)}, y^{(i)} \tag{5.7}$$

On the other hand, the temporal stream is computed through LK-OF proposed in [247]. This method involves the calculation of horizontal and vertical components at t as a derivative of dxt and dyt. The flow channels dx and yt of L form consecutive frames at $2L$ input channels to reduce the processing latency. Additionally, the Lucas-Kanade method assumes that the optical flow points v_x and v_y are constant within a small window W of size $n \times n$ pixels. Thus, the optical flow holds all pixels of the coordinates $q = (k, l)$ to window W as expressed in the equation follows:

$$I_x(q)v_x + I_y(q)v_y = -I_t(q) \quad \forall q = (k, l) \in W \tag{5.8}$$

Where I_x, I_y, and $I_t$are the partial derivative of the image intensity with respect to v_x, v_y. However, to obtain a compromise solution by the least squares principle, we computed a transpose matrix of A as in the equation follows:

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i)\, I_y(q_i) \\ \sum_i I_y(q_i)\, I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i)\, I_t(q_i) \\ -\sum_i I_y(q_i)\, I_t(q_i) \end{bmatrix} \tag{5.9}$$

This solves the 2×2 system, being computationally more efficient, and less sensitive to image noise than point-wise methods by assuming the flow is essentially constant in a local pixel neighborhood under consideration.

Overall, the design of both spatial and temporal model has much fewer parameters (≈ 1.3M) and one order of magnitude less computation overhead compared to all available alternatives [245] with a slight decrease in accuracy. Besides, the model is compressed by QAT (8-bit) and layer fusion for efficient implementation on edge SoC-FPGAs.

Accelerator Design

The proposed accelerator was built on top of Lin's project [244] and was further optimized for real-time performance. It leveraged the PL fabric to compute and parallelize the fused convolutional, batch-norm, ReLU, and max pooling layers alongside the Lukas-Kanade Optical Flow (LK-OF) separately on the PE and the SIMD. This approach allowed us to speed up the most intensive multiplication matrices by utilizing dedicated buffers within the memory, DSP, and LUT resources. On the other side, the remaining layers in the design (FC and Linear Fusion) were carried out on the PS part to complete the prediction action as illustrated in Fig. 35. This HW/SW scheme resulted in a reduction of the output data that is needed to be read from the DDR4 by shared buffers.

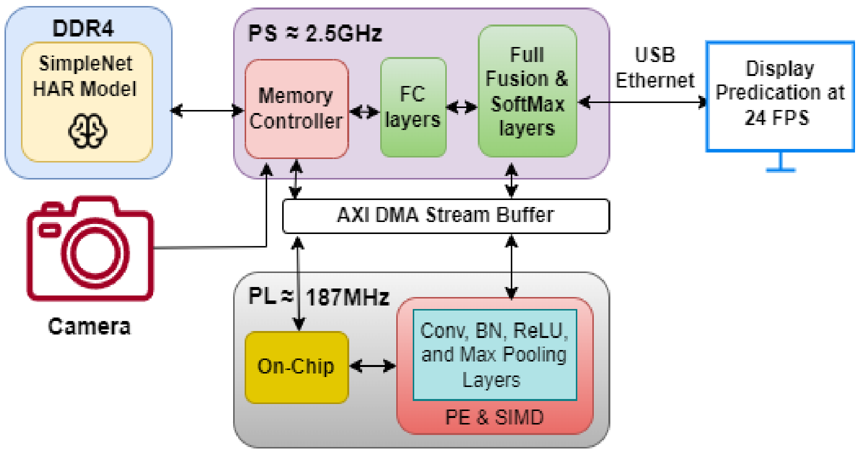

Figure 35. The mixed HW/SW accelerator for QHAR Layers on SoC-FPGA.

The developed accelerator incorporates a scheduling mechanism that efficiently transfers data between inputs, weights, and outputs using the high-performance AXI interconnect and memory controller. This is achieved through the effective execution of up-sizing, downsizing, and routing operations. The accelerator receives inputs from the processor's DDR4, executes the layers, and outputs the results back to the processor. The weights are then stored on-chip BRAM, allowing concurrent access by the partitioned PE elements in parallel, which leads to faster execution. This improves the external memory bandwidth and ensures simultaneous data read/write of IFM, OFM, and LK-OF from DDR4 to the on-chip engine and then from the on-chip engine to the compute unit. Lastly, equations of the FPS performed by the accelerator and the performance measurement in Giga Operations per second (GOP/s) are illustrated respectfully:

$$Frame\ Per\ Second\ (FPS) = \frac{No.of\ Frames\ processed}{Total\ time\ (second)} \tag{5.10}$$

$$Performance\ GOP/s = \frac{Total\ operation}{Excuation\ Time\ (second)} \times 10^9 \tag{5.11}$$

Loop Tiling Transformation

Loop tiling and loop unrolling are utilized as optimization techniques for the spatial network in our architecture, specifically for the ZCU104 board. This approach improves data locality and parallelizes the design. We fetched small portions of the data on BRAM by tiling the loop alongside the row, column, and channel directions of the IFM and OFM and converted them into a one-pixel channel. Then, the corresponding data in all the input channels were element-wise multiplied by filter weights and then summed together to save the resources and reduce the number of loop iterations, resulting in more efficient instruction scheduling.

Host Code

Python with PYNQ framework [35] is used to write the two-stream application driver-code that communicates with the hardware via Linux kernel, as shown in Fig. 36. The driver loads the accelerator bitstreams and model weights into the PL part, passes input/output buffers from DDR4, and run the prediction actions inference when deployed on USB camera or video files.

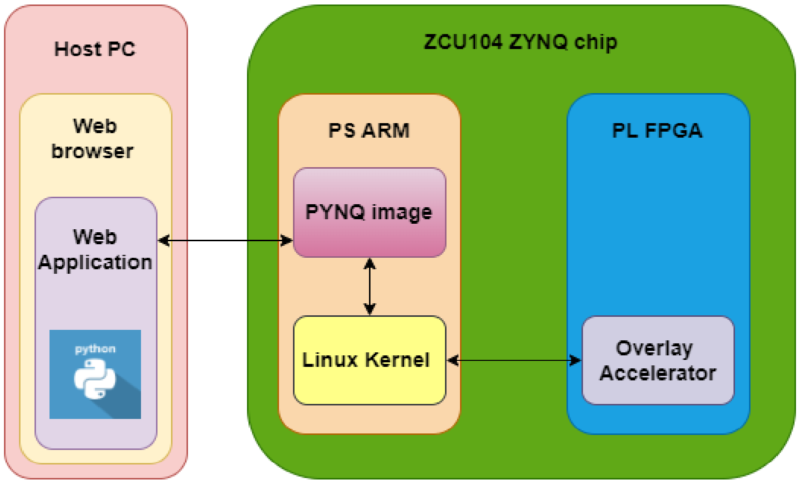

Figure 36. Software control of the accelerator via driver and operating system.

Experimental Setup and Results

Development Environments

The proposed two-stream QHAR SimpleNet was trained and tested on Google Colab powered by Nvidia A100 GPU with PyTorch framework and FX-Graph QAT library. Vivado HLS synthesized the accelerator bitstream and analyzed the available resources on ZCU104 board after place and route. Furthermore, experiments were conducted on two well-known action recognition datasets, UCF101 and UCF24, which contain 101 and 24 action classes, accordingly, and 13, 320, and 6500 video clips respectfully.

Results

Our QHAR architecture was trained from scratch on full precision and then fine-tuned (quantized weights) to an 8-bit unsigned integer. This approach achieved ≈79% and ≈81%

prediction accuracy on UCF101 and UCF24 datasets respectfully. The enhanced two-stream SimpleNet demonstrated higher accuracy results on smaller datasets with fewer classes. This is due to the fewer number of parameters (≈1.3M) and being lightweight architecture. On the other hand, the optimized accelerator operated at a frequency of 187MHz and achieved a performance of up to 120 GOP/s on ZCU104. Additionally, it was implemented with a USB camera demonstrating 22.5 FPS alongside the prediction classes as shown in Fig. 37. The throughput shots sometimes fluctuate in a range of (22 – 24.5) FPS when running the inference due to the model and hardware variability. We also ran the inference with video clips on ZCU104 and achieved more than 30 FPS as shown in Fig. 37. This phenomenon is due to the camera delay in real-time, being preprocessed and post-processed on the PS part. Nevertheless, the developed accelerator-based QHAR proved to be efficient in terms of resource usage, as shown in Table 9. Lastly, our network design and accelerator inference results are reported in Table 10 showing some comparisons and benchmarks of our QHAR model (SimNet) against the CPU, GPU, NVIDIA Jetson Nano, and the previous ResNet18-FPGA study.

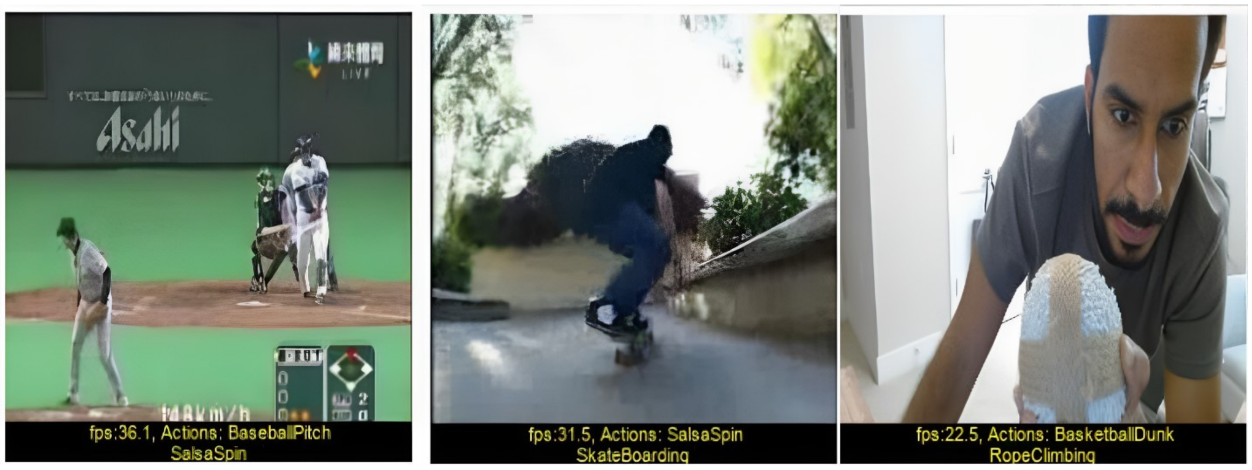
fps:36.1, Actions: BaseballPitch
SalsaSpin
fps:31.5, Actions: SalsaSpin
SkateBoarding
fps:22.5, Actions: BasketballDunk
RopeClimbing

Figure 37. Action classified from videos and USB camera shown the action name and No. of FPS.

Table 9. Resource utilization comparison

Device	**ZCU104 (our)**	**ZCU102 (prior study)**
Flip-Flops	83k (18%)	No report
LUTs	19.1k (38%)	227.8k (81%)
BRAM	22.6k (59%)	472 (13%)
DSP Slices	967 (59%)	1390 (54%)
Frequency	187 MHz	200 MHz

Table 10. Benchmarking with several platforms and studies

Platform	**GPU T4**	**CPU Intel Xeon ®**	**Jetson Nano**	**ZCU102 (previous study)**	**ZCU104 (our)**
Model	SimNet	SimNet	SimNet	ResNet18	SimNet
Dataset	UCF24	UCF24	UCF24	UCF101	UCF24
Bit width	8-bit	8-bit	8-bit	8-bit	8-bit
M-size	5.1 MB	5.1 MB	5.1 MB	22.3 MB	5.1 MB
Accuracy	81%	81%	81%	86%	81%
Frequency	585MHz	2.2 GHz	1.5GHz	200MHz	187MHz
GOP/s	-	-	-	4.12	120
FPS	≈51	≈1	≈9	≈15	≈24

Conclusion and Future Work

This research project introduced a scalable real-time QHAR-based hardware accelerator for two-stream CNN on SoC-FPGA. The proposed technique optimized SimpleNet with homogeneous layers and LK-OF estimation method for less computation and higher accuracy. As a result, a range of 1.7x - 1.9x higher throughput was achieved along with fewer network parameters compared to the previous research. This optimistic end-to-end open-source framework [248] can be more customized to work with different boards and datasets. Future work includes improving our architecture accuracy and developing an open-source multimodal real-time accelerator for Advanced Driver Assistance System with HAR that can recognize not only driver actions but also actions of pedestrians, cyclists, and other vehicles on the road, similar to Tesla Autopilot. This work was published in the 2023 IEEE 20th International Conference on Smart Communities: Improving Quality of Life using AI, Robotics and IoT (HONET) [61].

CHAPTER SIX: QUANTIZATION-BASED TWO-TEACHERS NET ON THE EDGE

Introduction

QAT and KD, shown in Fig. 38, have emerged as two prominent approaches for CNN compression. QAT is a technique that simulates the quantization process during training, allowing the model to adapt its weights and activations to the reduced precision [108]. Additionally, the weights and activations of the CNN are quantized to a lower precision representation, such as 8-bit integers, which reduces the memory footprint and computational complexity compared to the original 32-bit floating-point representation. By training the model with quantization in mind, QAT minimizes the accuracy loss caused by the quantization process, as the model learns to adjust its parameters to compensate for the reduced precision [111].

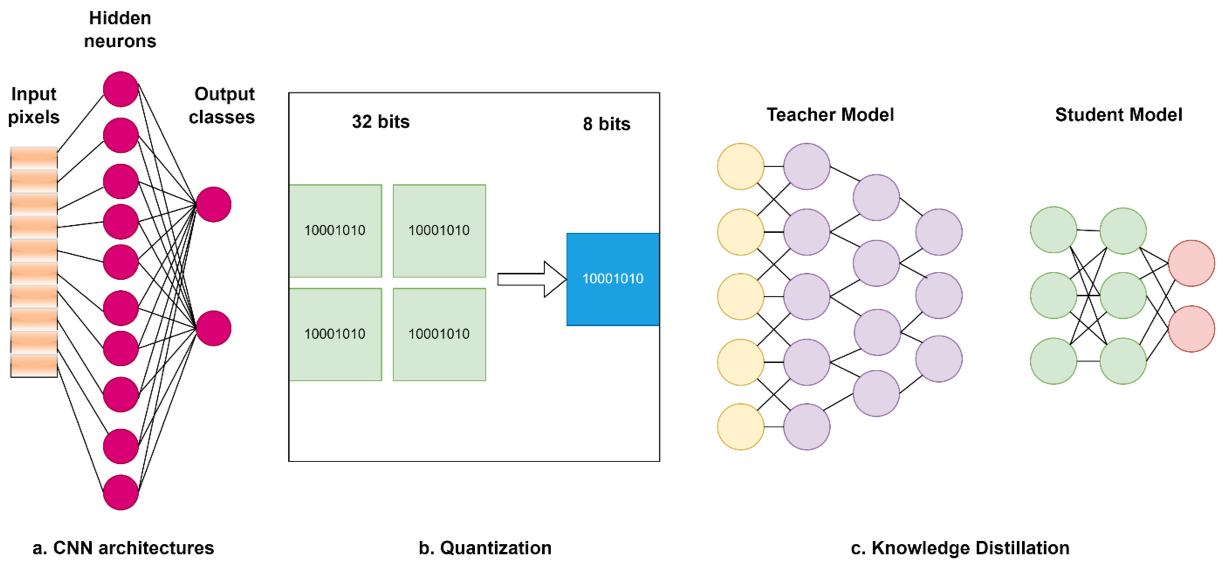

Figure 38. Description of combining quantization and knowledge distillation within one CNN.

On the other hand, KD is a technique that transfers knowledge from a large, pre-trained teacher model to a smaller student model [20]. The key idea behind KD is to leverage the knowledge captured by the teacher model, which has been trained on a large dataset and has achieved high performance, to guide the training of the student model. In KD, the student model learns to mimic the behavior of the teacher model by minimizing a loss function that measures the discrepancy between the outputs of the two models. By doing so, the student model can benefit from the teacher's knowledge and achieve comparable performance with reduced computational complexity.

Recent research has explored the combination of QAT and KD to achieve even higher compression rates and improved accuracy [251]. By integrating KD into the QAT process, the student model can learn from the teacher model's knowledge while being quantized, resulting in a compressed model that retains a significant portion of the teacher's performance. This combined approach has shown promising results in various CNN architectures and applications, such as image classification, object detection, and semantic segmentation. The process of combining QAT and KD typically involves the following steps:

1. Training a large, high-performance teacher model on the target task.
2. Applying QAT to the student model, simulating the quantization process during training.
3. Using the pre-trained teacher model to generate soft labels or hints for the training data.
4. Training the quantized student model using the standard loss function (e.g., cross-entropy) and a KD loss function (e.g., Kullback-Leibler divergence).
5. Fine-tune the quantized student model using the target dataset to improve its performance.

The benefits of combining QAT and KD for CNN compression are manifold:

1. Reduced Model Size: QAT allows for using lower-precision weights and activations, significantly reducing the model size compared to full-precision models.
2. Faster Inference: Quantized models require fewer computational resources and can be efficiently executed on hardware with limited precision support, leading to faster inference times.
3. Improved Accuracy: KD helps the student model learn from the teacher model's knowledge, resulting in higher accuracy than standalone quantization.
4. Flexibility: The combined approach can be applied to various CNN architectures and tasks, providing a versatile solution for model compression.
5. Energy Efficiency: Compressed models consume less energy during inference, making them suitable for deployment on battery-powered edge devices.

QAT and KD Challenges for Training and Testing

One of the critical challenges in combining QAT and KD is the selection of appropriate hyperparameters. The choice of quantization bit-width, distillation temperature, and loss function weights can significantly impact the performance of the compressed model. To address this challenge, researchers have proposed various techniques for hyperparameter optimization, such as Bayesian optimization and reinforcement learning. These techniques automate the search for optimal hyperparameters, reducing the manual effort required and improving the overall performance of the compressed model.

Another important aspect of combining QAT and KD is the choice of teacher and student models. The teacher model should be a high-performance model trained on a large dataset and achieve state-of-the-art accuracy. On the other hand, the student model should be a smaller model with reduced computational complexity and memory footprint. The selection of the student model architecture is crucial, as it determines the trade-off between compression rate and accuracy.

Related Work

The combination of QAT and KD has demonstrated its effectiveness in reducing model size, accelerating inference, and maintaining high accuracy. Mishra and Marr [251] proposed Apprentice, a framework that combines QAT and KD to train low-precision student models called "quantization distillation," where the student model learns from both the teacher model's outputs and the intermediate feature maps. On the ImageNet dataset, using ResNet-18 as the base model, Apprentice achieved a 68.5% top-1 accuracy with 4-bit quantization, resulting in a high compression rate compared to the full-precision model.

In addition, Zhang et al. [252] introduced Deep Mutual Learning (DML), a framework that combines QAT and KD where multiple student models are trained simultaneously, and they learn from each other through KD. On the CIFAR-100 dataset, using ResNet-32 as the base model, DML achieved a 70.6% top-1 accuracy with 4-bit quantization, just below the baseline full-precision model by 2.2%.

Chen et al. [253] developed a hardware-aware CNN compression framework called AdaDeep, which combines QAT, KD, and NAS. AdaDeep automatically designs efficient CNN architectures for edge devices, considering the target device’s hardware constraints and energy consumption. By jointly optimizing the quantization parameters, knowledge distillation process, and network architecture, AdaDeep achieved state-of-the-art performance on various edge devices. On the ImageNet dataset, using EfficientNet-B0 as the base model, AdaDeep achieved a 75.8% top-1 accuracy with 4-bit quantization, resulting in a 3.8x speedup on a Raspberry Pi device.

However, it is essential to note that the benchmark results may vary depending on the specific hardware platform, CNN architecture, and dataset used. Therefore, it is crucial to consider the experimental setup, evaluation metrics, and target devices when comparing different CNN compression methods to ensure a fair comparison. Furthermore, the choice of quantization bit-width, distillation technique, and compression pipeline may impact the final performance of the compressed model.

Objective

Compression techniques such as pruning, quantization, and KD are essential for deploying AI models on edge hardware using tools like Vitis-AI, Intel OpenVINO, Nvidia TensorRT, and ONNX, ensuring efficient real-time inference. However, these platforms require advanced methods like combining of two or three compression methods to maximize their potential. Those end-to-end frameworks are semi-open-source AI tools that can enhance interoperability, enabling the conversion and optimization of models for various hardware platforms like FPGAs, GPUs, CPU, and AI accelerators, ensuring efficient training and execution across diverse deployment environments for real-time inference. They address the challenges on resource constraints platforms by enabling lower bit-width operations and reducing model sizes, leading to faster computation on limited on-chip memory and logic resources, improving power efficiency, and improving throughput for handling real-time data streams effectively.

While Large Language Models (LLMs) [254] have garnered significant attention recently, CNNs remain a more suitable choice for edge computing applications. This is due CNNs are inherently more efficient regarding of memory and computational requirements, making them ideal for resource-constrained edge devices [4]. The growing focus on LLMs often overshadows the practical benefits of CNNs for edge computing. LLMs typically demand substantial computational power and memory, which can be impractical for edge devices. In contrast, CNNs can be efficiently quantized and pruned to fit the limited resources of edge hardware, making them more adaptable for real-time inference on devices with strict performance and power constraints.

The most recent work that aligns with this idea is proposed by Kim et al. [255], a method that coordinates quantization and KD in three phases: Self-studying, Co-studying, and Tutoring. This approach aims to mitigate and regulate the effect of KD on low-bit quantized models and improve the initialization and adaptability of the teacher network for better knowledge transfer. However, this method is different from our proposed idea. First, it utilized a single-teacher network and focused on making the teacher more quantization-friendly through joint training phases. Second, it implemented quantization through a uniform quantization scheme, which is hardware-friendly but might be efficient for some hardware types. Third, it demonstrated significant improvements in accuracy for low-bit quantized models but did not explore the use of multiple teachers for enhanced performance.

In this research project, we aim to develop an advanced compression technique that combines QAT and 2+KD to enhance the performance of CNNs on edge devices. This method utilized a dual-teacher knowledge distillation approach to achieve optimal class-specific accuracy in a quantized student model. The proposed technique addressed the current methods' limitations by ensuring compatibility with edge hardware, such as FPGAs and other edge accelerators, ensuring a scalable integration into various AI accelerator frameworks while maintaining high accuracy and efficiency. Our key contributions are as follows:

1. It employed two teacher networks to distill knowledge into the quantized INT 8-bit student model using cross-entropy loss, which calculate the optimal ratio of each class's knowledge from both teachers.
2. Integrated the PyTorch 2.0 FX-graph library, provided a flexible and robust framework for more fine-tuned control over the quantization process, ensuring compatibility with a broader range of edge hardware, including AMD Vitis-AI, Intel OpenVINO, NVIDIA TAO Toolkit, and ONNX frameworks.
3. Demonstrated the effect on image classification architectures such as EfficientNet [7], RegNet [256], and ConvNeXt [257] and datasets [Cifar-10 and Cifar-100] by 1% - 2% higher accuracy over existing solutions.

This technique helps resource constraints, improve power efficiency, reduce latency, and enhance throughput, ensuring that models can perform real-time inference effectively on various edge devices.

Methodology

Network Architecture

The "Two-Teachers Net" is a novel generic CNN compression technique designed to enhance the performance of quantized models on edge platforms. The teacher models are trained on the full-precision 32-bit weights and activations, while the student model is trained using QAT 8-bit with two teachers KD. The student model architecture is identical to the teacher models, with the exception of the quantized weights and activations. The training phase can be used with any CNN architecture and has four stages as shown in Fig. 39 to achieve a higher accuracy.

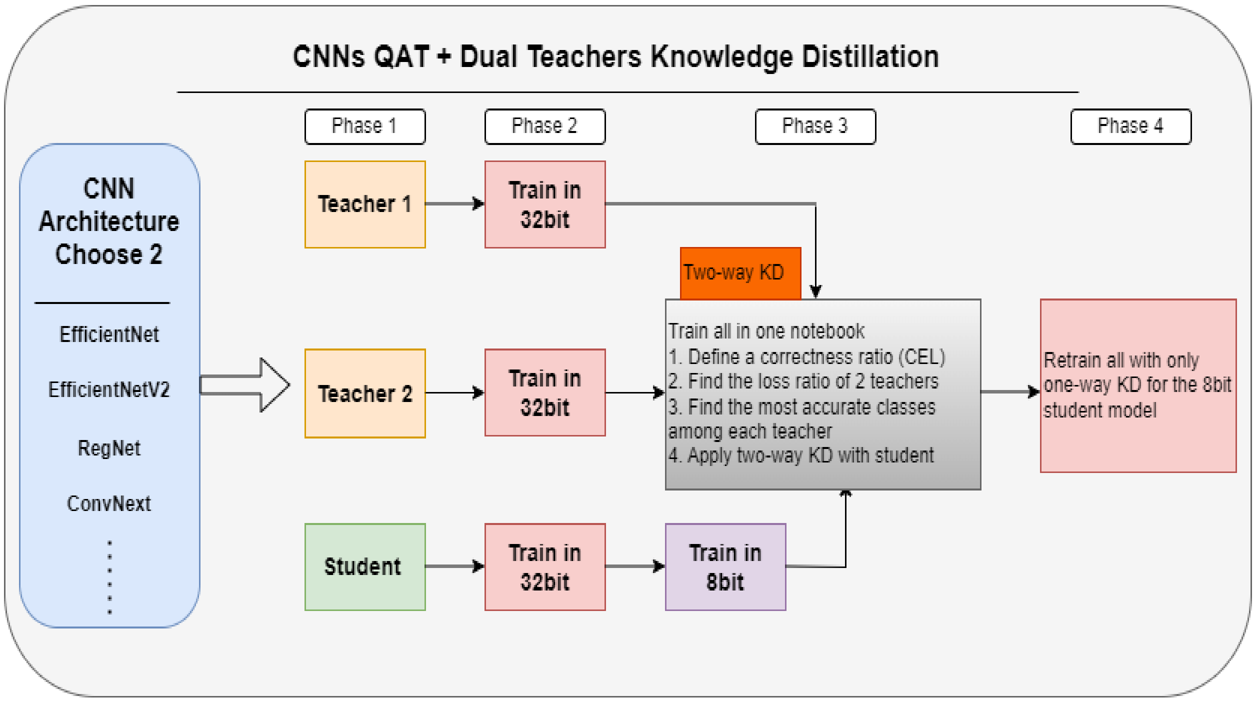
CNNs QAT + Dual Teachers Knowledge Distillation
Phase 1
Phase 2
Phase 3
Phase 4
CNN Architecture Choose 2
EfficientNet
EfficientNetV2
RegNet
ConvNext
Teacher 1
Train in 32bit
Teacher 2
Train in 32bit
Student
Train in 32bit
Train in 8bit
Two-way KD
Train all in one notebook
1. Define a correctness ratio (CEL)
2. Find the loss ratio of 2 teachers
3. Find the most accurate classes among each teacher
4. Apply two-way KD with student
Retrain all with only one-way KD for the 8bit student model

Figure 39. The proposed method "Two-Teacher Net". This method contains four stages to train two teachers models with one student model. Phase 1: choosing two CNN architectures (two large models for the two teachers, and of them should be a light version of one of them for image classification task. Phase 2: train all of them in full precision in separate notebook. Phase 3: train all of them in "two-way" one notebook by applying a modified equation through PyTorch. Phase 4: Train only the student in 8-bit in "one-way".

The knowledge distillation process is achieved through a combination of two loss functions:

- Cross-entropy loss (L_{CE}) between student model's output and the labels.
- Knowledge distillation loss (L_{KD}) between student model's and teachers' models' outputs.

The modified cross-entropy loss function (L_{mod}) is calculated as follows:

$$L_{mod} = L_{CE} + \alpha * L_{KD} \tag{6.1}$$

Where α is a hyperparameter that controls the strength of the knowledge distillation loss. To compute the two teachers' knowledge distillation best classes ratio and transfer them to the student model, we denoted the output of the two teachers' models as $T1$ and $T2$, respectively. Then, we computed the similarity between the two teachers' models using the following equation:

$$S = \frac{1}{N} \sum_{i=1}^{N} (T_1^i * T_2^i) \tag{6.2}$$

Where N is the number of classes, and $T_1^i \; and \; T_2^i$ are the output probabilities of the i^{th} class from the two teachers' models, respectively. In addition, the best classes ratio R can be computed as follows:

$$R = \frac{S}{\max(S)} \tag{6.3}$$

Where $\max(S)$ is the maximum similarity value across all classes. Next, we transferred the knowledge from the two teachers' models to the student model using the following equation:

$$L_{kd} = \sum_{i=1}^{N} (R_i * (T_1^i \log(S_i) + T_2^i \log(S_i))) \tag{6.4}$$

Where L_{kd} is the knowledge distillation loss, R_i is the best classes ratio for the i^{th} class, $T_1^i \ and \ T_2^i$ are the output probabilities of the i^{th} class from the two teachers' models, respectively, and S_i is the softmax output of the student model for the i^{th} class which can be calculated as follow:

$$S_i = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)} \tag{6.5}$$

Where S_i is the softmax output of the student model for the i^{th} class, $z_i \ and \ z_j$ are both the output of the student model, and N is the number of classes. Here, the denominator z_j is the sum of the exponentials of the outputs of the student model for all classes, which normalizes the softmax output to ensure it sums to 1. The final loss function for the student model is a combination of the cross-entropy loss and the knowledge distillation loss, as described in Eq. 6.1.

On the other hand, the student model has to be quantized to have lower computing calculation through the following equation:

$$w_q = round\left(\frac{w}{s_w}\right) * s_w \quad (6.6) \qquad , \qquad a_q = round\left(\frac{a}{s_a}\right) * s_a \quad (6.7)$$

Where w and a are the original weights and activations, s_w and s_a are the scaling factors, and w_q and a_q are the quantized weights and activations. Moreover, a straight-through estimator (STE) is needed to calculation backpropagation through quantization. It can be computed as in the equation follows:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{x}} = \frac{\partial \mathrm{L}}{\partial x_q} \tag{6.8}$$

Where x represents the model parameters (weights or activations) and x_q their quantized counterparts.

By following this comprehensive methodology, the student model learns to distill the knowledge from the two teachers' mode and can effectively be compressed and optimized through EfficientNet, RegNet, and ConvNeXt models, as shown in Fig. 40 for deployment on edge hardware, achieving high accuracy by almost 1% - 2% depending on the batch-size selected and number of epochs trained for. It is proven to be efficient and suitable for real-time image classification applications.

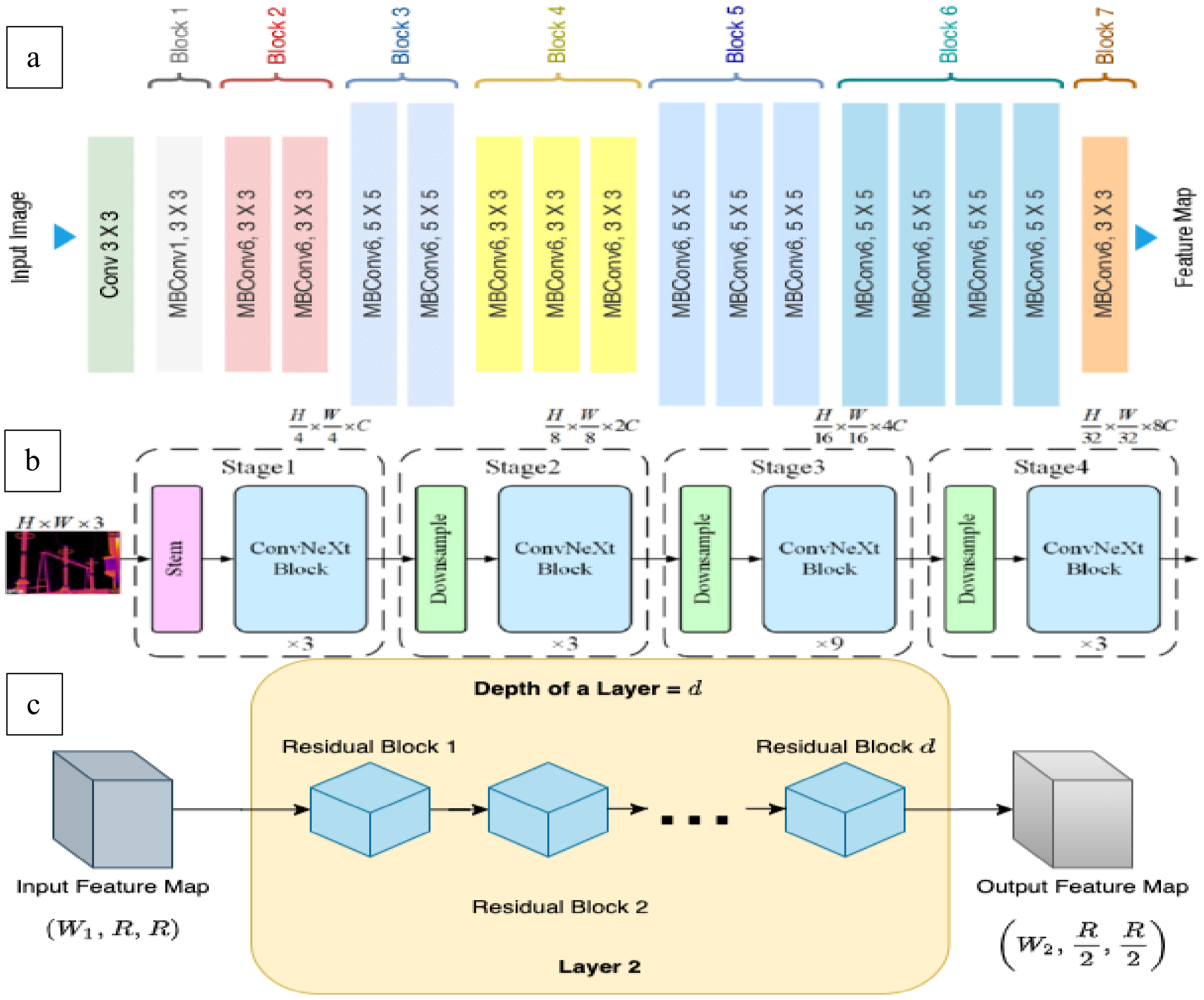

Figure 40. The architectures of a. EfficinetNet, b. RegNet, and c. ConvNeXt.

Framework Integration

The Two-Teachers Net compression method is a framework-agnostic approach that enables seamless integration into various AI frameworks and tools, making it a versatile solution for deploying efficient CNNs on edge devices. The key technical points that contribute to the research are:

1. The method's native implementation in PyTorch leverages the PyTorch 2.0 FX-graph mode for efficient 8-bit quantization of the student model, reducing memory footprint and computational complexity.
2. The trained and quantized student model is exported to the ONNX format using PyTorch's torch.onnx.export function [258] to integrate with frameworks and tools like Vitis-AI, OpenVINO, TensorRT, and ONNX.
3. The exported ONNX model can be imported into the target AI framework or tool for further optimization and deployment.

Experimental Setup and Results

Development Environments

The proposed Two-Teachers Net compression method was trained and tested on Google Colab powered by Nvidia A100 GPU with PyTorch framework and FX-Graph QAT library. With ONNX conversion, we ensured interoperability and broad hardware compatibility by converting models into this standard format. Furthermore, experiments were conducted on three architectures, EfficientNet, RegNet, and ConvNeXt, and tested with two well-known image classification datasets, Cifar-10 and Cifar-100.

Results

Our method was trained from scratch with four stages on full precision, as shown in Fig. 1, and then fine-tuned (quantized weights and activation) to an 8-bit unsigned integer for the student model. This approach achieved almost higher accuracy on Ciar-10 and Cifar-100 datasets by 1% - 2% over existing solutions. The enhanced Two-Teachers Net was trained with 50 epochs for Cifar-10 and 30 epochs for Cifar-100 and demonstrated higher accuracy results on RegNet and ConvNeXt smallest models, respectfully. These lightweight architectures are indeed needed for edge deployment on CPU or FPGA. We have trained these models with four stages and shown the accuracy and losses for each stage. All of them were trained with the EfficinetNet large model as the main teacher baseline model. Nevertheless, the developed method proved to be efficient and provided high accuracy with fewer number of trained epochs, as shown in Table 11.

Table 11. The accuracy results when comparing two architectures keeping in that EfficinetNet was always the teachers one model for both of RegNet and ConvNeXt.

Architectures	**RegNet**		**ConvNeXt**	
Dataset	Cifar-100		Cifar-10	
No. of training epochs	30 - 50		30 - 50	
Metric measurement	Accuracy	Loss	Accuracy	Loss
Phase 1 - Student model 32-bit	78%	≈ 1.5	85%	≈ 0.25
Phase 2 - Student model 8-bit	95%	≈ 0.2	97.65%	≈ 0.2
Phase 3 – Two-way: 2-KD & student 8-bit	95.7%	≈ 0.1	97.8%	≈ 0.13
Phase 4 – One-way: 2-KD & student 8-bit	97.5%	≈ 0.05	99.1%	≈ 0.08

Conclusion and Future Work

This research project introduced an efficient dual-teacher QAT compression method for edge inference. The proposed technique optimized different CNN architectures with homogeneous and parallel inline-four training phases for less computation and fewer number of epochs. As a result, a range of 1% - 2% higher accuracy was achieved alongside the capability of integration into various AI frameworks. This optimistic end-to-end open-source framework can be more customized to work with different architectures and datasets. Future work includes improving accuracy and integrating this solution for medical imaging low-cost real-time hardware for hospitals and clinics that can run AI medical imaging for patients to recognize several diseases, similar to the MONAI project. This work has been submitted to the 2024 IEEE 21st International Conference on Smart Communities: Improving Quality of Life using AI, Robotics and IoT (HONET).

CHAPTER SEVEN: CONCLUSION AND FUTURE WORK

In the era of AI applications, where there is a significant need for processing capacity, clouds and datacenters have emerged as the primary locations for data processing. As the demand scaling and Moore's law is slowing down where multi-core processors approach their limits, heterogeneous architectures and hardware accelerators for AI and DNN are becoming increasingly important to meet the growing demands for processing these algorithms. FPGAs have attracted the interest of researchers and industry as AI hardware accelerators due to their flexibility, reconfigurability, energy efficiency, and potential for high performance for edge computing and on-device AI data processing.

However, due to the distinct programming nature of FPGAs compared to software programming, application designers find them challenging to employ. Unlike other peripherals, FPGAs are hardware devices that may be reconfigured. Interfacing with them is more complex since particular communication methods are required for each design. In order to utilize FPGA accelerators, it is necessary to have an interface framework that can integrate applications running on the host with the FPGA accelerators. Vitis-AI and IntelOpenVINO are AI training and inference accelerators for AMD and Intel FPGAs for many applications that can execute Linux machines. Unlike power-hungry hardware such as GPUs, FPGAs are power-efficient and can be customized based on the algorithm's layers for high performance.

This dissertation focuses on two parts: CNN compression for improving the efficiency and accuracy of CNNs and FPGAs HLS accelerators designed for real-time inference. Chapter 3 introduced a transformative contribution to the FINN project by partitioning a novel HW/SW co-design approaches that optimize the critical trade-off between throughput and resource utilization. We have developed a cutting-edge, scalable streaming dataflow architecture for training quantized deconvolution GAN (QDCGAN) on FPGAs, harnessing the power of an efficient deconvolution engine with paralleled parallelism. This accelerator design has been augmented by an end-to-end open-source framework that streamlines the entire training, implementation, state-space exploration, and inference scaling process using Vivado HLS for AMD SoC-FPGAs. By providing a complete toolchain for GAN acceleration on edge devices, our work has broken new ground in deploying generative CNN-based models in resource-constrained environments, unlocking many new applications in domains such as image synthesis, data augmentation, and anomaly detection.

In Chapter 4, we have introduced a highly efficient and scalable SoC-FPGA CNN-based accelerator design optimized of performance and resource utilization. Our innovative template leverages advanced techniques like loop tiling transformation and dataflow modeling to convert convolutional and fully connected layers into vector multiplication between input and output feature maps. This results in a single, highly optimized compute unit on-chip. By analyzing the computational workload, data dependency, and external memory bandwidth, our accelerator has achieved a peak performance of 230 GOP/s on ZYNQ boards, setting a new standard for CNN acceleration on edge devices. This groundbreaking work has opened a new implication for the deployment of high-performance computer vision models in various applications, from autonomous vehicles and robotics to intelligent surveillance systems.

In Chapter 5, we addressed the unique challenges of HAR on edge devices where we have developed an end-to-end HAR scalable HW/SW accelerator co-design based on an enhanced 8-bit quantized Two-Stream SimpleNet-PyTorch CNN architecture. Our approach fused convolutional, batch-norm, and ReLU operations into a homogeneous layer. It leveraged the cutting-edge Lucas-Kanade motion flow method to enable a high parallelism accelerator design optimized for on-chip engine computing. Our accelerator has shattered previous benchmarks by achieving an exceptional 81% prediction accuracy with an approximately 24 FPS real-time inference throughput at 187MHz on ZCU104, outperforming prior research by an astonishing 1.7x - 1.9x. This work set a new boundary in deploying HAR models in real-time surveillance, robotics, and human-computer interaction applications on edge devices.

In Chapter 6, this research has introduced a novel CNN compression technique called "Two-Teachers Net," which utilizes PyTorch FX-graph mode to train an 8-bit quantized student model using knowledge distillation from two teacher models. This innovative method improves the accuracy of the compressed model by 1%-2% compared to existing solutions for edge platforms. It can seamlessly integrated into AI hardware and software optimization toolchains without architectural adjustments. By advancing the state-of-the-art in model compression, our work has opened up new possibilities for deploying high-accuracy CNNs on low-power edge devices, enabling a wide range of applications in fields such as healthcare, smart cities, and industrial automation.

Future Work

One particularly promising direction for future work is the design of efficient and scalable HLS accelerators for Vision in Transformer (ViT) models, which have recently gained significant attention due to their exceptional performance in natural language processing and vision tasks, such as image classification, detection, segmentation without a specific annotated data or labels. Designing HLS accelerators for ViT presents a unique set of challenges due to their immense size and complexity. ViT such as DINOv2 [259], SAM (Segment Anything) [260], and EfficientViT [261], have billions of parameters and require substantial computational resources and memory bandwidth, making their deployment on ZYNQ SoC-FPGA and resource-constrained edge devices a challenging task.

To address these challenges, we should develop more optimized compression methods and create a novel HW/SW co-design approach that optimize the trade-off between performance, power consumption, and resource utilization on ZYNQ architecture while exploiting the inherent parallelism and sparsity of ViTs. One promising approach to designing efficient HLS accelerators for ViTs is to leverage the principles of dataflow architecture and fine-grained parallelism. By decomposing the ViTs into smaller, more manageable sub-modules, such as self-attention layers, feed-forward networks, and embedding layers, researchers can develop specialized hardware accelerators that exploit the unique characteristics of each sub-module. For example, the self-attention mechanism, which is a core component of a transformer, can be accelerated using a systolic array architecture that enables efficient matrix multiplication and data reuse. Similarly, the feed-forward networks can be accelerated using a combination of parallel processing elements and on-chip memory hierarchies that minimize data movement and maximize data locality.

To further optimize the performance and efficiency of HLS accelerators for ViTs, researchers should investigate advanced quantization techniques that reduce the bit-width of weights and activations without compromising model accuracy. One example of such a model is the one proposed by Microsoft (BitNet-model [262]) leveraging newer quantization and mixed-precision method results in 1.58-bit only for the entire model, where different sub-modules are quantized only to a minimum of 4-bit due to sensitivity errors. This can significantly reduce the memory footprint and computational complexity while maintaining high accuracy. Another critical aspect of designing efficient HLS accelerators for ViTs is the development of flexible and scalable architectures that can adapt to different ViTs configurations and target platforms. Future research should focus on creating modular and parameterizable HLS templates that allow for rapidly exploring different hardware architectures and design trade-offs.

To facilitate the deployment and inference of ViTs on FPGA, future research should also investigate the integration of HLS accelerators with existing AI frameworks and toolchains, such as TensorFlow, PyTorch, and ONNX. By developing standardized interfaces and APIs that allow for the seamless integration of HLS accelerators with these frameworks, researchers can enable the rapid development and deployment of ViTs-based applications on FPGA. Finally, the challenges of real-time inference for ViT on edge devices are critical at the current hardware limitation. By developing efficient scheduling algorithms and memory management techniques that enable the dynamic allocation of hardware resources based on the current workload and environmental conditions, researchers can ensure that ViTs can adapt to changing input data and user requirements in real-time.

In conclusion, designing efficient and scalable HLS accelerators for ViTs is a complex and multifaceted challenge that requires a holistic approach spanning hardware architecture, software algorithms, and deployment frameworks. By leveraging state-of-the-art techniques in dataflow architecture, compression, HW-SW co-design, and real-time inference, researchers can unlock the full potential of ViTs on FPGA and enable a new generation of intelligent, responsive, and adaptive edge computing systems. Future work in this area has the potential to revolutionize the way we interact with and benefit from AI technologies, ultimately leading to more efficient, effective, and user-friendly edge computing solutions that can transform a wide range of industries and applications.

LIST OF REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv Neural Inf Process Syst*, vol. 25, 2012, Accessed: Mar. 07, 2024. [Online]. Available: http://code.google.com/p/cuda-convnet/

[2] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature 2015 521:7553*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/nature14539.

[3] J. Wang, B. Cao, P. Yu, L. Sun, W. Bao, and X. Zhu, “Deep learning towards mobile applications,” *Proc Int Conf Distrib Comput Syst*, vol. 2018-July, pp. 1385–1393, Jul. 2018, doi: 10.1109/ICDCS.2018.00139.

[4] M. Tan and Q. V Le, “EfficientNetV2: Smaller Models and Faster Training.” PMLR, pp. 10096–10106, Jul. 01, 2021. Accessed: Mar. 09, 2024. [Online]. Available: https://proceedings.mlr.press/v139/tan21a.html

[5] “ImageNet.” Accessed: Mar. 09, 2024. [Online]. Available: https://www.image-net.org/update-mar-11-2021.php

[6] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Oct. 2015, Accessed: Mar. 07, 2024. [Online]. Available: https://arxiv.org/abs/1510.00149v5

[7] M. Tan and Q. V Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.” PMLR, pp. 6105–6114, May 24, 2019. Accessed: Mar. 09, 2024. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html

[8] X. Zhang, Y. Wang, and W. Shi, “{pCAMP}: Performance Comparison of Machine Learning Packages on the Edges.” 2018.

[9] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H. J. Yoo, “7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16,” *Dig Tech Pap IEEE Int Solid State Circuits Conf*, vol. 2019-February, pp. 142–144, Mar. 2019, doi: 10.1109/ISSCC.2019.8662302.

[10] A. Howard *et al.*, “Searching for MobileNetV3,” *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-October, pp. 1314–1324, May 2019, doi: 10.1109/ICCV.2019.00140.

[11] A. Dosovitskiy *et al.*, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” *ICLR 2021 - 9th International Conference on Learning Representations*, Oct. 2020, Accessed: Jun. 15, 2024. [Online]. Available: https://arxiv.org/abs/2010.11929v2

[12] N. Ma, X. Zhang, H. T. Zheng, and J. Sun, “ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design,” *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11218 LNCS, pp. 122–138, Jul. 2018, doi: 10.1007/978-3-030-01264-9_8.

[13] "CIFAR-10 Benchmark (Image Classification) | Papers With Code." Accessed: Jun. 15, 2024. [Online]. Available: https://paperswithcode.com/sota/image-classification-on-cifar-10

[14] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," Oct. 2017, Accessed: Mar. 07, 2024. [Online]. Available: https://arxiv.org/abs/1710.09282v9

[15] A. Nechi, L. Groth, S. Mulhem, F. Merchant, R. Buchty, and M. Berekovic, "FPGA-based Deep Learning Inference Accelerators: Where Are We Standing?," *ACM Trans Reconfigurable Technol Syst*, vol. 16, no. 4, pp. 1–32, Dec. 2023, doi: 10.1145/3613963.

[16] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Network," *Adv Neural Inf Process Syst*, vol. 28, 2015.

[17] Y. Cai, W. Hua, H. Chen, G. E. Suh, C. De Sa, and Z. Zhang, "Structured Pruning is All You Need for Pruning CNNs at Initialization," Mar. 2022, Accessed: Mar. 10, 2024. [Online]. Available: https://arxiv.org/abs/2203.02549v2

[18] Y. Guo, "A Survey on Methods and Theories of Quantized Neural Networks," Aug. 2018, Accessed: Mar. 10, 2024. [Online]. Available: https://arxiv.org/abs/1808.04752v2

[19] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization Through Weight Equalization and Bias Correction." pp. 1325–1334, 2019.

[20] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," Mar. 2015, Accessed: Mar. 10, 2024. [Online]. Available: https://arxiv.org/abs/1503.02531v1

[21] S. Yun, J. Park, K. Lee, and J. Shin, "Regularizing Class-Wise Predictions via Self-Knowledge Distillation." pp. 13876–13885, 2020. Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/alinlab/cs-kd.

[22] S. Jang, W. Liu, and Y. Cho, "Convolutional Neural Network Model Compression Method for Software—Hardware Co-Design," *Information 2022, Vol. 13, Page 451*, vol. 13, no. 10, p. 451, Sep. 2022, doi: 10.3390/INFO13100451.

[23] X. Zhang, Y. Wang, and W. Shi, "{pCAMP}: Performance Comparison of Machine Learning Packages on the Edges." 2018.

[24] G. Verma, Y. Gupta, A. M. Malik, and B. Chapman, "Performance Evaluation of Deep Learning Compilers for Edge Inference," *2021 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2021 - In conjunction with IEEE IPDPS 2021*, pp. 858–865, Jun. 2021, doi: 10.1109/IPDPSW52791.2021.00128.

[25] "Jetson Benchmarks | NVIDIA Developer." Accessed: Mar. 10, 2024. [Online]. Available: https://developer.nvidia.com/embedded/jetson-benchmarks

[26] P. Joshi, M. Hasanuzzaman, C. Thapa, H. Afli, and T. Scully, “Enabling All In-Edge Deep Learning: A Literature Review,” *IEEE Access*, vol. 11, pp. 3431–3460, 2023, doi: 10.1109/ACCESS.2023.3234761.

[27] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” *Proc Int Symp Comput Archit*, vol. Part F128643, pp. 1–12, Jun. 2017, doi: 10.1145/3079856.3080246.

[28] B. Moons, D. Bankman, L. Yang, B. Murmann, and M. Verhelst, “BinarEye: An always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28nm CMOS,” *2018 IEEE Custom Integrated Circuits Conference, CICC 2018*, pp. 1–4, May 2018, doi: 10.1109/CICC.2018.8357071.

[29] K. Guo, S. Zeng, J. Yu, Y. U. Wang, H. Yang, and Y. Wang, “A Survey of FPGA-Based Neural Network Accelerator,” *ACM Trans. Reconng. Technol. Syst*, vol. 9, no. 11, Dec. 2017, Accessed: Mar. 10, 2024. [Online]. Available: https://arxiv.org/abs/1712.08934v3

[30] A. R. Garola *et al.*, “A Zynq-Based Flexible ADC Architecture Combining Real-Time Data Streaming and Transient Recording,” *IEEE Trans Nucl Sci*, vol. 68, no. 2, pp. 245–249, Feb. 2021, doi: 10.1109/TNS.2020.3035146.

[31] S. Mittal, “A survey of FPGA-based accelerators for convolutional neural networks,” *Neural Comput Appl*, vol. 32, no. 4, pp. 1109–1139, Feb. 2020, doi: 10.1007/S00521-018-3761-1/FIGURES/29.

[32] S. I. Venieris, A. Kouris, and C. S. Bouganis, “Toolflows for Mapping Convolutional Neural Networks on FPGAs,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, Jun. 2018, doi: 10.1145/3186332.

[33] X. Zhang *et al.*, “DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs,” *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, Nov. 2018, doi: 10.1145/3240765.3240801.

[34] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGAs: From prototyping to deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011, doi: 10.1109/TCAD.2011.2110592.

[35] “PYNQ Introduction — Python productivity for Zynq (Pynq).” Accessed: May 18, 2024. [Online]. Available: https://pynq.readthedocs.io/en/latest/

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

[37] A. Sateesan, S. Sinha, S. K. G, and A. P. Vinod, “A Survey of Algorithmic and Hardware Optimization Techniques for Vision Convolutional Neural Networks on FPGAs,” *Neural*

Process Lett, vol. 53, no. 3, pp. 2331–2377, Jun. 2021, doi: 10.1007/S11063-021-10458-1/TABLES/10.

[38] S. I. Venieris and C. S. Bouganis, "FpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," *Proceedings - 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016*, pp. 40–47, Aug. 2016, doi: 10.1109/FCCM.2016.22.

[39] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A Survey," May 2018, Accessed: Mar. 11, 2024. [Online]. Available: https://arxiv.org/abs/1806.01683v1

[40] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks," 2020, doi: 10.1007/978-3-031-01766-7.

[41] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, pp. 243–254, Feb. 2016, doi: 10.1109/ISCA.2016.30.

[42] "Zynq UltraScale+ MPSoC Embedded Design Tutorial — Embedded Design Tutorials 2022.1 documentation." Accessed: Apr. 17, 2024. [Online]. Available: https://xilinx.github.io/Embedded-Design-Tutorials/docs/2022.1/build/html/docs/Introduction/ZynqMPSoC-EDT/ZynqMPSoC-EDT.html

[43] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-Based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019, doi: 10.1109/ACCESS.2018.2890150.

[44] "Zynq UltraScale+ MPSoC." Accessed: Mar. 10, 2024. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[45] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," *FPGA 2015 - 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, Feb. 2015, doi: 10.1145/2684746.2689060.

[46] M. Sun *et al.*, "FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks with Intra-Layer, Mixed-Precision Quantization," *FPGA 2022 - Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 134–145, Feb. 2022, doi: 10.1145/3490422.3502364.

[47] Z. Shao *et al.*, "Memory-Efficient CNN Accelerator Based on Interlayer Feature Map Compression," Oct. 2021, Accessed: Apr. 17, 2024. [Online]. Available: https://arxiv.org/abs/2110.06155v1

[48] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, Dec. 2016, doi: 10.1145/3020078.3021744.

[49] "Welcome to hls4ml's documentation! — hls4ml 0.8.1 documentation." Accessed: Apr. 14, 2024. [Online]. Available: https://fastmachinelearning.org/hls4ml/

[50] S. Jang, W. Liu, and Y. Cho, "Convolutional Neural Network Model Compression Method for Software—Hardware Co-Design," *Information 2022, Vol. 13, Page 451*, vol. 13, no. 10, p. 451, Sep. 2022, doi: 10.3390/INFO13100451.

[51] "FINN | finn." Accessed: Mar. 17, 2024. [Online]. Available: https://xilinx.github.io/finn/

[52] "GitHub - fastmachinelearning/hls4ml: Machine learning on FPGAs using HLS." Accessed: Mar. 17, 2024. [Online]. Available: https://github.com/fastmachinelearning/hls4ml

[53] "Home - Ultralytics YOLOv8 Docs." Accessed: Mar. 27, 2024. [Online]. Available: https://docs.ultralytics.com/

[54] I. Goodfellow *et al.*, "Generative adversarial networks," *Commun ACM*, vol. 63, no. 11, pp. 139–144, Oct. 2020, doi: 10.1145/3422622.

[55] "Deep Learning Super Sampling (DLSS) | NVIDIA Developer." Accessed: Mar. 18, 2024. [Online]. Available: https://developer.nvidia.com/rtx/dlss

[56] A. Alhussain and M. Lin, "Hardware-Efficient Deconvolution-Based GAN for Edge Computing," *2022 56th Annual Conference on Information Sciences and Systems, CISS 2022*, pp. 172–176, 2022, doi: 10.1109/CISS53076.2022.9751185.

[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Adv Neural Inf Process Syst*, vol. 25, 2012, Accessed: Mar. 18, 2024. [Online]. Available: http://code.google.com/p/cuda-convnet/

[58] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, Sep. 2014, Accessed: Mar. 18, 2024. [Online]. Available: https://arxiv.org/abs/1409.1556v6

[59] A. Alhussain and M. Lin, "Hardware-Efficient Template-Based Deep CNNs Accelerator Design," *2022 IEEE International Conference on Networking, Architecture and Storage, NAS 2022 - Proceedings*, 2022, doi: 10.1109/NAS55553.2022.9925552.

[60] S. H. Hasanpour, M. Rouhani, M. Fayyaz, and M. Sabokrou, "Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures," Aug. 2016, Accessed: Mar. 18, 2024. [Online]. Available: https://arxiv.org/abs/1608.06037v8

[61] A. Alhussain and M. Lin, "FPGA-QHAR: Throughput-Optimized for Quantized Two-Stream Human Action Recognition on the Edge," *2023 IEEE 20th International Conference on Smart Communities: Improving Quality of Life using AI, Robotics and IoT, HONET 2023*, pp. 156–160, 2023, doi: 10.1109/HONET59747.2023.10374852.

[62] "Vitis AI — Vitis^{TM} AI 3.5 documentation." Accessed: Mar. 17, 2024. [Online]. Available: https://xilinx.github.io/Vitis-AI/3.5/html/index.html

[63] "OpenVINO 2024 — OpenVINO^{TM} documentation — Version(2024)." Accessed: Mar. 17, 2024. [Online]. Available: https://docs.openvino.ai/2024/home.html

[64] “GitHub - NVIDIA/TensorRT: NVIDIA® TensorRT^{TM} is an SDK for high-performance deep learning inference on NVIDIA GPUs. This repository contains the open source components of TensorRT.” Accessed: Mar. 17, 2024. [Online]. Available: https://github.com/NVIDIA/TensorRT

[65] “GitHub - onnx/onnx: Open standard for machine learning interoperability.” Accessed: Mar. 17, 2024. [Online]. Available: https://github.com/onnx/onnx

[66] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, “Accelerating CNN inference on FPGAs: A Survey,” May 2018, Accessed: Mar. 11, 2024. [Online]. Available: https://arxiv.org/abs/1806.01683v1

[67] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature 2015 521:7553*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/nature14539.

[68] D. P. Kingma and J. L. Ba, “Adam: A Method for Stochastic Optimization,” *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, Dec. 2014, Accessed: Mar. 24, 2024. [Online]. Available: https://arxiv.org/abs/1412.6980v9

[69] A. M. Fred Agarap, “Deep Learning using Rectified Linear Units (ReLU),” Mar. 2018, Accessed: Mar. 24, 2024. [Online]. Available: https://arxiv.org/abs/1803.08375v2

[70] S. Sathyanarayana, “A Gentle Introduction to Backpropagation,” 2014, Accessed: Mar. 25, 2024. [Online]. Available: www.numericinsight.com

[71] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition.” pp. 770–778, 2016. Accessed: Mar. 24, 2024. [Online]. Available: http://image-net.org/challenges/LSVRC/2015/

[72] L. Liu, X. Liu, J. Gao, W. Chen, and J. Han, “Understanding the Difficulty of Training Transformers,” *EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pp. 5747–5763, Apr. 2020, doi: 10.18653/v1/2020.emnlp-main.463.

[73] M. Abadi *et al.*, *{TensorFlow}: A System for {Large-Scale} Machine Learning*. 2016. Accessed: Mar. 25, 2024. [Online]. Available: https://tensorflow.org.

[74] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” *Adv Neural Inf Process Syst*, vol. 32, 2019.

[75] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, “A survey on deep transfer learning,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11141 LNCS, pp. 270–279, 2018, doi: 10.1007/978-3-030-01424-7_27/COVER.

[76] C. Shorten and T. M. Khoshgoftaar, “A survey on Image Data Augmentation for Deep Learning,” *J Big Data*, vol. 6, no. 1, pp. 1–48, Dec. 2019, doi: 10.1186/S40537-019-0197-0/FIGURES/33.

[77] "COCO - Common Objects in Context." Accessed: Mar. 25, 2024. [Online]. Available: https://cocodataset.org/#home

[78] "OmniVec: Learning robust representations with cross modal sharing | Papers With Code." Accessed: Mar. 25, 2024. [Online]. Available: https://paperswithcode.com/paper/omnivec-learning-robust-representations-with

[79] "CIFAR-10 and CIFAR-100 datasets." Accessed: Mar. 25, 2024. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[80] "The PASCAL Visual Object Classes Homepage." Accessed: Mar. 28, 2024. [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/

[81] "Tiny ImageNet | Kaggle." Accessed: Mar. 28, 2024. [Online]. Available: https://www.kaggle.com/competitions/tiny-imagenet/overview

[82] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding Batch Normalization," *Adv Neural Inf Process Syst*, vol. 31, 2018.

[83] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6354 LNCS, no. PART 3, pp. 92–101, 2010, doi: 10.1007/978-3-642-15825-4_10/COVER.

[84] C. M. Bishop, "PATTERN RECOGNITION AND MACHINE LEARNING CHAPTER 8: GRAPHICAL MODELS Part I".

[85] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks." pp. 4510–4520, 2018.

[86] P. Mattson *et al.*, "MLPerf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, Mar. 2020, doi: 10.1109/MM.2020.2974843.

[87] V. J. Reddi *et al.*, "MLPerf Inference Benchmark," *Proc Int Symp Comput Archit*, vol. 2020-May, pp. 446–459, May 2020, doi: 10.1109/ISCA45697.2020.00045.

[88] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An Overview on Edge Computing Research," *IEEE Access*, vol. 8, pp. 85714–85728, 2020, doi: 10.1109/ACCESS.2020.2991734.

[89] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 779–788, Jun. 2015, doi: 10.1109/CVPR.2016.91.

[90] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "CSPNet: A New Backbone That Can Enhance Learning Capability of CNN." pp. 390–391, 2020.

[91] "Segment - Ultralytics YOLOv8 Docs." Accessed: Mar. 28, 2024. [Online]. Available: https://docs.ultralytics.com/tasks/segment/#models

[92] M. Hu, Y. Li, L. Fang, and S. Wang, "A2-FPN: Attention Aggregation Based Feature Pyramid Network for Instance Segmentation." pp. 15343–15352, 2021.

[93] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11141 LNCS, pp. 270–279, 2018, doi: 10.1007/978-3-030-01424-7_27/COVER.

[94] P. Ren *et al.*, "A Comprehensive Survey of Neural Architecture Search," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, p. 76, May 2021, doi: 10.1145/3447582.

[95] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," *Adv Neural Inf Process Syst*, vol. 2015-January, pp. 1135–1143, Jun. 2015, Accessed: Mar. 29, 2024. [Online]. Available: https://arxiv.org/abs/1506.02626v3

[96] "Neural Network Pruning: A Gentle Introduction | by SoonChang | Medium." Accessed: Jun. 18, 2024. [Online]. Available: https://pohsoonchang.medium.com/neural-network-pruning-update-cda56343e5a2

[97] H. Li, H. Samet, A. Kadav, I. Durdanovic, and H. P. Graf, "Pruning Filters for Efficient ConvNets," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Aug. 2016, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/1608.08710v3

[98] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-October, pp. 1398–1406, Jul. 2017, doi: 10.1109/ICCV.2017.155.

[99] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning Structured Sparsity in Deep Neural Networks," *Adv Neural Inf Process Syst*, pp. 2082–2090, Aug. 2016, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/1608.03665v4

[100] X. Gao, Y. Zhao, L. Dudziak, R. Mullins, and X. Cheng-Zhong, "Dynamic Channel Pruning: Feature Boosting and Suppression," *7th International Conference on Learning Representations, ICLR 2019*, Oct. 2018, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/1810.05331v2

[101] T. J. Yang, Y. H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 6071–6079, Nov. 2016, doi: 10.1109/CVPR.2017.643.

[102] J. Lin, W. M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny Deep Learning on IoT Devices," *Adv Neural Inf Process Syst*, vol. 2020-December, Jul. 2020, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/2007.10319v2

[103] S. Vadera and S. Ameen, "Methods for Pruning Deep Neural Networks," *IEEE Access*, vol. 10, pp. 63280–63300, 2022, doi: 10.1109/ACCESS.2022.3182659.

[104] J. Frankle and M. Carbin, “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks,” *7th International Conference on Learning Representations, ICLR 2019*, Mar. 2018, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/1803.03635v5

[105] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning Structured Sparsity in Deep Neural Networks,” *Adv Neural Inf Process Syst*, pp. 2082–2090, Aug. 2016, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/1608.03665v4

[106] X. Dong, S. Chen, and S. J. Pan, “Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon,” *Adv Neural Inf Process Syst*, vol. 2017-December, pp. 4858–4868, May 2017, Accessed: Mar. 30, 2024. [Online]. Available: https://arxiv.org/abs/1705.07565v2

[107] J. Lin, Y. Rao, J. Lu, and J. Zhou, “Runtime Neural Pruning,” *Adv Neural Inf Process Syst*, vol. 30, 2017.

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

[109] Y. Bengio, N. Léonard, and A. Courville, “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation,” Aug. 2013, Accessed: Mar. 31, 2024. [Online]. Available: https://arxiv.org/abs/1308.3432v1

[110] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” Jun. 2018, Accessed: Mar. 31, 2024. [Online]. Available: https://arxiv.org/abs/1806.08342v1

[111] S. Wu, G. Li, F. Chen, and L. Shi, “Training and Inference with Integers in Deep Neural Networks,” *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Feb. 2018, Accessed: Mar. 31, 2024. [Online]. Available: https://arxiv.org/abs/1802.04680v1

[112] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “HAQ: Hardware-Aware Automated Quantization with Mixed Precision,” *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 8604–8612, Nov. 2018, doi: 10.1109/CVPR.2019.00881.

[113] “(prototype) FX Graph Mode Quantization User Guide — PyTorch Tutorials 2.2.1+cu121 documentation.” Accessed: Mar. 31, 2024. [Online]. Available: https://pytorch.org/tutorials/prototype/fx_graph_mode_quant_guide.html

[114] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” Mar. 2015, Accessed: Mar. 11, 2024. [Online]. Available: https://arxiv.org/abs/1503.02531v1

[115] A. Alkhulaifi, F. Alsahli, and I. Ahmad, “Knowledge distillation in deep learning and its applications,” *PeerJ Comput Sci*, vol. 7, pp. 1–24, Apr. 2021, doi: 10.7717/PEERJ-CS.474/FIG-5.

[116] T. Furlanello, Z. C. Lipton, M. Tschannen, L. Itti, and A. Anandkumar, “Born Again Neural Networks,” *35th International Conference on Machine Learning, ICML 2018*, vol. 4, pp. 2615–2624, May 2018, Accessed: Mar. 31, 2024. [Online]. Available: https://arxiv.org/abs/1805.04770v2

[117] A. Malinin, B. Mlodozeniec, and M. Gales, “Ensemble Distribution Distillation,” *8th International Conference on Learning Representations, ICLR 2020*, Apr. 2019, Accessed: Mar. 31, 2024. [Online]. Available: https://arxiv.org/abs/1905.00076v3

[118] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Feb. 2018, Accessed: Mar. 31, 2024. [Online]. Available: https://arxiv.org/abs/1802.05668v1

[119] T. Li, J. Li, Z. Liu, and C. Zhang, “Few Sample Knowledge Distillation for Efficient Network Compression”, Accessed: Mar. 31, 2024. [Online]. Available: http://sites.google.com/view/icml2019-on-device-compact-dnn

[120] F. Tung and G. Mori, “Similarity-Preserving Knowledge Distillation,” *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-October, pp. 1365–1374, Jul. 2019, doi: 10.1109/ICCV.2019.00145.

[121] A. Trivedi, T. Udagawa, M. Merler, R. Panda, Y. El-Kurdi, and B. Bhattacharjee, “Neural Architecture Search for Effective Teacher-Student Knowledge Transfer in Language Models,” Mar. 2023, Accessed: Apr. 01, 2024. [Online]. Available: https://arxiv.org/abs/2303.09639v2

[122] J. Yim, D. Joo, J. Bae, and J. Kim, “A Gift from Knowledge Distillation: Fast Optimization, Network Minimization and Transfer Learning”.

[123] S. Shen *et al.*, “Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT,” *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence*, pp. 8815–8824, Sep. 2019, doi: 10.1609/aaai.v34i05.6409.

[124] S. I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh, “Improved Knowledge Distillation via Teacher Assistant,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, pp. 5191–5198, Apr. 2020, doi: 10.1609/AAAI.V34I04.5963.

[125] M. Jeon *et al.*, *Analysis of {Large-Scale} {Multi-Tenant} {GPU} Clusters for {DNN} Training Workloads*. 2019. Accessed: Apr. 05, 2024. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/dice

[126] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge Computing: Vision and Challenges,” *IEEE Internet Things J*, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[127] “GitHub - mlcommons/inference: Reference implementations of MLPerf^{TM} inference benchmarks.” Accessed: Apr. 07, 2024. [Online]. Available: https://github.com/mlcommons/inference

[128] "MLCommons · GitHub." Accessed: Apr. 07, 2024. [Online]. Available: https://github.com/mlcommons

[129] ChenTianshi *et al.*, "DianNao," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, Feb. 2014, doi: 10.1145/2654822.2541967.

[130] ChenYu-Hsin, EmerJoel, and SzeVivienne, "Eyeriss," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, Jun. 2016, doi: 10.1145/3007787.3001177.

[131] S. Venkataramani *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," *Proc Int Symp Comput Archit*, vol. Part F128643, pp. 13–26, Jun. 2017, doi: 10.1145/3079856.3080244.

[132] C. Ding *et al.*, "CIRCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. Part F131207, pp. 395–408, Oct. 2017, doi: 10.1145/3123939.3124552.

[133] "CUDA Deep Neural Network (cuDNN) | NVIDIA Developer." Accessed: Apr. 06, 2024. [Online]. Available: https://developer.nvidia.com/cudnn

[134] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, "Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product," *Proceedings - Symposium on Computer Arithmetic*, vol. 2020-June, pp. 133–136, Jun. 2020, doi: 10.1109/ARITH48897.2020.00029.

[135] K. Guo *et al.*, "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, Jan. 2018, doi: 10.1109/TCAD.2017.2705069.

[136] X. Zhang *et al.*, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, Nov. 2018, doi: 10.1145/3240765.3240801.

[137] R. Nane *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016, doi: 10.1109/TCAD.2015.2513673.

[138] "Zynq UltraScale+ MPSoC - Xilinx Wiki - Confluence." Accessed: Apr. 05, 2024. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/444006775/Zynq+UltraScale+MPSoC

[139] "Vivado Design Suite User Guide: Partial Reconfiguration • Viewer • AMD Technical Information Portal." Accessed: Apr. 05, 2024. [Online]. Available: https://docs.amd.com/v/u/2020.1-English/ug909-vivado-partial-reconfiguration

[140] "Introduction — Embedded Design Tutorials 2021.2 documentation." Accessed: Jun. 30, 2024. [Online]. Available: https://xilinx.github.io/Embedded-Design-Tutorials/docs/2021.2/build/html/docs/User_Guides/SPA-UG/docs/1-introduction.html

[141] "Zynq® UltraScale+^{TM} MPSoC Data Sheet: Overview (DS891) • Viewer • AMD Technical Information Portal." Accessed: Apr. 06, 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview

[142] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," Apr. 2017, Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/1704.04861v1

[143] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 1800–1807, Oct. 2016, doi: 10.1109/CVPR.2017.195.

[144] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated Residual Transformations for Deep Neural Networks," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 5987–5995, Nov. 2016, doi: 10.1109/CVPR.2017.634.

[145] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 6848–6856, Jul. 2017, doi: 10.1109/CVPR.2018.00716.

[146] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning Structured Sparsity in Deep Neural Networks," *Adv Neural Inf Process Syst*, pp. 2082–2090, Aug. 2016, Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/1608.03665v4

[147] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-October, pp. 1398–1406, Jul. 2017, doi: 10.1109/ICCV.2017.155.

[148] H. Li, H. Samet, A. Kadav, I. Durdanovic, and H. P. Graf, "Pruning Filters for Efficient ConvNets," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Aug. 2016, Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/1608.08710v3

[149] A. Koratana, D. Kang, P. Bailis, and M. Zaharia, "LIT: Block-wise Intermediate Representation Training for Model Compression," Oct. 2018, Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/1810.01937v1

[150] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9908 LNCS, pp. 525–542, Mar. 2016, doi: 10.1007/978-3-319-46493-0_32.

[151] B. Liu, F. Li, X. Wang, B. Zhang, and J. Yan, "Ternary Weight Networks," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2023-June, May 2016, doi: 10.1109/ICASSP49357.2023.10094626.

[152] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, Feb. 2015, Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/1502.03167v3

[153] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster,” *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 326–331, Aug. 2016, doi: 10.1145/2934583.2934644.

[154] Y. Ma, Y. Cao, S. Vrudhula, and J. S. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 45–54, Feb. 2017, doi: 10.1145/3020078.3021736.

[155] “Vivado Design Suite: AXI Reference Guide (UG1037) • Viewer • AMD Technical Information Portal.” Accessed: Apr. 05, 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1037-vivado-axi-reference-guide

[156] M. Courbariaux, Y. Bengio, and J. P. David, “BinaryConnect: Training Deep Neural Networks with binary weights during propagations,” *Adv Neural Inf Process Syst*, vol. 2015-January, pp. 3123–3131, Nov. 2015, Accessed: Apr. 10, 2024. [Online]. Available: https://arxiv.org/abs/1511.00363v3

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

[158] “7 Series FPGAs Memory Resources User Guide (UG473) • Viewer • AMD Technical Information Portal.” Accessed: Apr. 11, 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/ug473_7Series_Memory_Resources

[159] J. Qiu *et al.*, “Going deeper with embedded FPGA platform for convolutional neural network,” *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, Feb. 2016, doi: 10.1145/2847263.2847265.

[160] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, Nov. 2019, doi: 10.1109/TCAD.2017.2785257.

[161] R. Nane *et al.*, “A Survey and Evaluation of FPGA High-Level Synthesis Tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016, doi: 10.1109/TCAD.2015.2513673.

[162] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer CNN accelerators,” *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2016-December, Dec. 2016, doi: 10.1109/MICRO.2016.7783725.

[163] Y. Ma, Y. Cao, S. Vrudhula, and J. S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, Oct. 2017, doi: 10.23919/FPL.2017.8056824.

[164] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN Accelerator Efficiency Through Resource Partitioning," *Proc Int Symp Comput Archit*, vol. Part F128643, pp. 535–547, Jun. 2016, doi: 10.1145/3079856.3080221.

[165] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J. sun Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration*, vol. 62, pp. 14–23, Jun. 2018, doi: 10.1016/J.VLSI.2017.12.009.

[166] T. Yang *et al.*, "BISWSRBS: A Winograd-based CNN Accelerator with a Fine-grained Regular Sparsity Pattern and Mixed Precision Quantization," *ACM Trans Reconfigurable Technol Syst*, vol. 14, no. 4, Sep. 2021, doi: 10.1145/3467476.

[167] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *ACM SIGPLAN Notices*, vol. 26, no. 6, pp. 30–44, May 1991, doi: 10.1145/113446.113449.

[168] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 553–564, May 2017, doi: 10.1109/HPCA.2017.29.

[169] S. I. Venieris and C. S. Bouganis, "Latency-driven design for FPGA-based convolutional neural networks," *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, Oct. 2017, doi: 10.23919/FPL.2017.8056828.

[170] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of CNN training on deeply-pipelined FPGA clusters with work and weight load balancing," *Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018*, pp. 394–398, Nov. 2018, doi: 10.1109/FPL.2018.00074.

[171] S. H. Chua, T. H. Teo, M. A. Tiruye, and I. C. Wey, "Systolic Array Based Convolutional Neural Network Inference on FPGA," *Proceedings - 2022 IEEE 15th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2022*, pp. 128–133, 2022, doi: 10.1109/MCSOC57363.2022.00029.

[172] X. Wei *et al.*, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," *Proc Des Autom Conf*, vol. Part 128280, Jun. 2017, doi: 10.1145/3061639.3062207.

[173] Y. Li and Y. Du, "A Novel Software-Defined Convolutional Neural Networks Accelerator," *IEEE Access*, vol. 7, pp. 177922–177931, 2019, doi: 10.1109/ACCESS.2019.2956841.

[174] R. S. Molina, V. Gil-Costa, M. L. Crespo, and G. Ramponi, "High-Level Synthesis Hardware Design for FPGA-Based Accelerators: Models, Methodologies, and Frameworks," *IEEE Access*, vol. 10, pp. 90429–90455, 2022, doi: 10.1109/ACCESS.2022.3201107.

[175] W. Jiang *et al.*, “Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search,” *Proc Des Autom Conf*, Jan. 2019, doi: 10.1145/3316781.3317757.

[176] A. Misra and V. Kindratenko, “HLS-Based Acceleration Framework for Deep Convolutional Neural Networks,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12083 LNCS, pp. 221–231, 2020, doi: 10.1007/978-3-030-44534-8_17/TABLES/2.

[177] Y. Wang, H. Li, and X.-W. Li, “DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family Error Tolerant Computing View project HALO for elastic IoT View project,” 2016, doi: 10.1145/2897937.2898003.

[178] A. Rahman, S. Oh, J. Lee, and K. Choi, “Design Space Exploration of FPGA Accelerators for Convolutional Neural Networks”, doi: 10.5555/3130379.3130650.

[179] K. Bjerge, J. H. Schougaard, and D. E. Larsen, “A scalable and efficient convolutional neural network accelerator using HLS for a system-on-chip design,” *Microprocess Microsyst*, vol. 87, p. 104363, Nov. 2021, doi: 10.1016/J.MICPRO.2021.104363.

[180] “Introduction • Vitis High-Level Synthesis User Guide (UG1399) • Reader • AMD Technical Information Portal.” Accessed: Apr. 14, 2024. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls

[181] Y. Umuroglu, D. Conficconi, L. Rasnayake, T. B. Preusser, and M. Sjalander, “Optimizing Bit-Serial Matrix Multiplication for Reconfigurable Computing,” *ACM Trans Reconfigurable Technol Syst*, vol. 12, no. 3, p. 24, Jan. 2019, doi: 10.1145/3337929.

[182] J. Duarte *et al.*, “Fast inference of deep neural networks in FPGAs for particle physics,” *Journal of Instrumentation*, vol. 13, no. 7, Apr. 2018, doi: 10.1088/1748-0221/13/07/P07027.

[183] J. Ngadiuba *et al.*, “Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml,” *Mach Learn Sci Technol*, vol. 2, no. 1, p. 015001, Dec. 2020, doi: 10.1088/2632-2153/ABA042.

[184] J. Chen and X. Ran, “Deep Learning with Edge Computing: A Review,” *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019, doi: 10.1109/JPROC.2019.2921977.

[185] J. Cong, P. Wei, C. H. Yu, and P. Zhang, “Automated accelerator generation and optimization with composable, parallel and pipeline architecture,” *Proc Des Autom Conf*, vol. Part F137710, Jun. 2018, doi: 10.1145/3195970.3195999.

[186] “Intel® oneAPI Base Toolkit: Essential oneAPI Tools & Libraries.” Accessed: Apr. 16, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html#gs.82tcna

[187] “TAO Toolkit | NVIDIA Developer.” Accessed: Apr. 15, 2024. [Online]. Available: https://developer.nvidia.com/tao-toolkit

[188] "Intel® oneAPI Programming Guide." Accessed: Apr. 16, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2024-1/overview.html

[189] "Performance Benchmarks — OpenVINO^{TM} documentation — Version(2024)." Accessed: Apr. 15, 2024. [Online]. Available: https://docs.openvino.ai/2024/about-openvino/performance-benchmarks.html

[190] "TAO Toolkit Quick Start Guide - NVIDIA Docs." Accessed: Apr. 15, 2024. [Online]. Available: https://docs.nvidia.com/tao/tao-toolkit/text/tao_toolkit_quick_start_guide.html

[191] "Developer Guide :: NVIDIA Deep Learning TensorRT Documentation." Accessed: Apr. 15, 2024. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html

[192] "ONNC/onnc: Open Neural Network Compiler." Accessed: Apr. 16, 2024. [Online]. Available: https://github.com/ONNC/onnc

[193] "Apache MXNet | A flexible and efficient library for deep learning." Accessed: Apr. 15, 2024. [Online]. Available: https://mxnet.apache.org/versions/1.9.1/

[194] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved Techniques for Training GANs," *Adv Neural Inf Process Syst*, pp. 2234–2242, Jun. 2016, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1606.03498v1

[195] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Nov. 2015, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1511.06434v2

[196] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," Jan. 2017, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1701.07875v3

[197] T. Karras, S. Laine, and T. Aila, "A Style-Based Generator Architecture for Generative Adversarial Networks," *IEEE Trans Pattern Anal Mach Intell*, vol. 43, no. 12, pp. 4217–4228, Dec. 2018, doi: 10.1109/TPAMI.2020.2970919.

[198] S. Nowozin, B. Cseke, and R. Tomioka, "f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization," *Adv Neural Inf Process Syst*, pp. 271–279, Jun. 2016, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1606.00709v1

[199] M. Arjovsky and L. Bottou, "Towards Principled Methods for Training Generative Adversarial Networks," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Jan. 2017, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1701.04862v1

[200] A. Brock, J. Donahue, and K. Simonyan, "Large Scale GAN Training for High Fidelity Natural Image Synthesis," *7th International Conference on Learning Representations, ICLR 2019*, Sep. 2018, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1809.11096v2

[201] L. Theis, A. Van Den Oord, and M. Bethge, “A note on the evaluation of generative models,” *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Nov. 2015, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1511.01844v3

[202] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium,” *Adv Neural Inf Process Syst*, vol. 2017-December, pp. 6627–6638, Jun. 2017, doi: 10.18034/ajase.v8i1.9.

[203] Z. Wang, Q. She, and T. E. Ward, “Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy,” *ACM Comput Surv*, vol. 54, no. 2, Jun. 2019, doi: 10.1145/3439723.

[204] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, “Self-Attention Generative Adversarial Networks,” *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 12744–12753, May 2018, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/1805.08318v2

[205] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, “Convergence of Edge Computing and Deep Learning: A Comprehensive Survey,” *IEEE Communications Surveys and Tutorials*, vol. 22, no. 2, pp. 869–904, Apr. 2020, doi: 10.1109/COMST.2020.2970550.

[206] M. Li, J. Lin, Y. Ding, Z. Liu, J.-Y. Zhu, and S. Han, “GAN Compression: Efficient Architectures for Interactive Conditional GANs,” *IEEE Trans Pattern Anal Mach Intell*, vol. 44, no. 12, pp. 9331–9346, Mar. 2020, doi: 10.1109/TPAMI.2021.3126742.

[207] N. Shrivastava, M. A. Hanif, S. Mittal, S. R. Sarangi, and M. Shafique, “A survey of hardware architectures for generative adversarial networks,” *Journal of Systems Architecture*, vol. 118, p. 102227, Sep. 2021, doi: 10.1016/J.SYSARC.2021.102227.

[208] A. Zhou *et al.*, “Hardware-Aware Graph Neural Network Automated Design for Edge Computing Platforms,” Mar. 2023, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/2303.10875v2

[209] N. Shrivastava, M. A. Hanif, S. Mittal, S. R. Sarangi, and M. Shafique, “A survey of hardware architectures for generative adversarial networks,” *Journal of Systems Architecture*, vol. 118, p. 102227, Sep. 2021, doi: 10.1016/J.SYSARC.2021.102227.

[210] A. Yazdanbakhsh *et al.*, “FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks,” *Proceedings - 26th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018*, pp. 65–72, Sep. 2018, doi: 10.1109/FCCM.2018.00019.

[211] C. Ledig *et al.*, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network,” *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 105–114, Sep. 2016, doi: 10.1109/CVPR.2017.19.

[212] V. Bhatia and Y. Kumar, “Attaining Real-Time Super-Resolution for Microscopic Images Using GAN,” Oct. 2020, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/2010.04634v1

[213] I. Colbert, J. Daly, K. Kreutz-Delgado, and S. Das, “A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Limited Edge Computing Applications?,” Jan. 2021, Accessed: May 17, 2024. [Online]. Available: https://arxiv.org/abs/2102.00294v2

[214] A. Alhussain and M. Lin, “Hardware-Efficient Deconvolution-Based GAN for Edge Computing,” *2022 56th Annual Conference on Information Sciences and Systems, CISS 2022*, pp. 172–176, 2022, doi: 10.1109/CISS53076.2022.9751185.

[215] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, “Improved Training of Wasserstein GANs,” *Adv Neural Inf Process Syst*, vol. 2017-December, pp. 5768–5778, Mar. 2017, Accessed: May 18, 2024. [Online]. Available: https://arxiv.org/abs/1704.00028v3

[216] “GitHub - Xilinx/finn-hlslib: Vitis HLS Library for FINN.” Accessed: May 18, 2024. [Online]. Available: https://github.com/Xilinx/finn-hlslib

[217] “GitHub - Xilinx/brevitas: Brevitas: neural network quantization in PyTorch.” Accessed: May 18, 2024. [Online]. Available: https://github.com/Xilinx/brevitas

[218] S. H. Khan, M. Hayat, and F. Porikli, “Regularization of deep neural networks with spectral dropout,” *Neural Networks*, vol. 110, pp. 82–90, Feb. 2019, doi: 10.1016/J.NEUNET.2018.09.009.

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

[220] A. T. Y. Chen, M. Biglari-Abhari, K. I. K. Wang, A. Bouzerdoum, and F. H. C. Tivive, “Convolutional neural network acceleration with hardware/software co-design,” *Applied Intelligence*, vol. 48, no. 5, pp. 1288–1301, May 2018, doi: 10.1007/s10489-017-1007-z.

[221] S. Mittal and S. Vaishay, “A survey of techniques for optimizing deep learning on GPUs,” *Journal of Systems Architecture*, vol. 99, p. 101635, Oct. 2019, doi: 10.1016/J.SYSARC.2019.101635.

[222] H. Kimm, I. Paik, and H. Kimm, “Performance Comparision of TPU, GPU, CPU on Google Colaboratory over Distributed Deep Learning,” *Proceedings - 2021 IEEE 14th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2021*, pp. 312–319, 2021, doi: 10.1109/MCSOC51149.2021.00053.

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

[224] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, “Comparing energy Efficiency of CPU, GPU and FPGA implementations for vision kernels,” *2019 IEEE*

International Conference on Embedded Software and Systems, ICESS 2019, Jun. 2019, doi: 10.1109/ICESS.2019.8782524.

[225] A. Alhussain and M. Lin, "Hardware-Efficient Template-Based Deep CNNs Accelerator Design," *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–4, Oct. 2022, doi: 10.1109/NAS55553.2022.9925552.

[226] S. Ji, W. Xu, M. Yang, and K. Yu, "3D Convolutional neural networks for human action recognition," *IEEE Trans Pattern Anal Mach Intell*, vol. 35, no. 1, pp. 221–231, 2013, doi: 10.1109/TPAMI.2012.59.

[227] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks".

[228] J. Carreira, A. Zisserman, Z. Com, and † Deepmind, "Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 4724–4733, May 2017, doi: 10.1109/CVPR.2017.502.

[229] D. Tran, H. Wang, L. Torresani, J. Ray, Y. Lecun, and M. Paluri, "A Closer Look at Spatiotemporal Convolutions for Action Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 6450–6459, Nov. 2017, doi: 10.1109/CVPR.2018.00675.

[230] S. Yan, Y. Xiong, and D. Lin, "Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 7444–7452, Jan. 2018, Accessed: May 31, 2024. [Online]. Available: https://arxiv.org/abs/1801.07455v2

[231] L. Shi, Y. Zhang, J. Cheng, and H. Lu, "Two-Stream Adaptive Graph Convolutional Networks for Skeleton-Based Action Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 12018–12027, May 2018, doi: 10.1109/CVPR.2019.01230.

[232] K. Simonyan and A. Zisserman, "Two-Stream Convolutional Networks for Action Recognition in Videos," *Adv Neural Inf Process Syst*, vol. 1, no. January, pp. 568–576, Jun. 2014, Accessed: Jun. 18, 2024. [Online]. Available: https://arxiv.org/abs/1406.2199v2

[233] C. Zach, T. Pock, and H. Bischof, "A Duality Based Approach for Realtime TV-L 1 Optical Flow," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4713 LNCS, pp. 214–223, 2007, doi: 10.1007/978-3-540-74936-3_22.

[234] L. Wang *et al.*, "Temporal Segment Networks: Towards Good Practices for Deep Action Recognition," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9912 LNCS, pp. 20–36, Aug. 2016, doi: 10.1007/978-3-319-46484-8_2.

[235] H. B. Zhang *et al.*, "A Comprehensive Survey of Vision-Based Human Action Recognition Methods," *Sensors 2019, Vol. 19, Page 1005*, vol. 19, no. 5, p. 1005, Feb. 2019, doi: 10.3390/S19051005.

[236] M. Al-Faris, J. Chiverton, D. Ndzi, and A. I. Ahmed, "A Review on Computer Vision-Based Methods for Human Action Recognition," *J Imaging*, vol. 6, no. 6, Jun. 2020, doi: 10.3390/JIMAGING6060046.

[237] A. Dosovitskiy *et al.*, "FlowNet: Learning Optical Flow With Convolutional Networks." pp. 2758–2766, 2015.

[238] T. W. Hui, X. Tang, and C. C. Loy, "LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 8981–8989, May 2018, doi: 10.1109/CVPR.2018.00936.

[239] "Kinetics Dataset | Papers With Code." Accessed: Jun. 01, 2024. [Online]. Available: https://paperswithcode.com/dataset/kinetics

[240] "CRCV | Center for Research in Computer Vision at the University of Central Florida." Accessed: Jun. 01, 2024. [Online]. Available: https://www.crcv.ucf.edu/data/UCF101.php

[241] S. Nooruddin, M. M. Islam, and F. Karray, "TinyHAR: Benchmarking Human Activity Recognition Systems in Resource Constrained Devices," *2022 IEEE 8th World Forum on Internet of Things, WF-IoT 2022*, 2022, doi: 10.1109/WF-IOT54382.2022.10152039.

[242] A. Sarabu and A. K. Santra, "Distinct Two-Stream Convolutional Networks for Human Action Recognition in Videos Using Segment-Based Temporal Modeling," *Data 2020, Vol. 5, Page 104*, vol. 5, no. 4, p. 104, Nov. 2020, doi: 10.3390/DATA5040104.

[243] Y. Sun and A. M. Kist, "Deep Learning on Edge TPUs," Aug. 2021, Accessed: Jun. 01, 2024. [Online]. Available: https://arxiv.org/abs/2108.13732v2

[244] J. M. Lin, K. T. Lai, B. R. Wu, and M. S. Chen, "Efficient two-stream action recognition on FPGA," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 3070–3074, Jun. 2021, doi: 10.1109/CVPRW53098.2021.00343.

[245] S. H. Hasanpour, M. Rouhani, M. Fayyaz, and M. Sabokrou, "Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures," Aug. 2016, Accessed: Jun. 01, 2024. [Online]. Available: https://github.com/Coderx7/SimpleNet_Pytorch

[246] P. Girdhar, P. Johri, and D. Virmani, "An Improved Empirical Hyper-Parameter Tuned Supervised Model for Human Activity Recognition based on Motion Flow and Deep Learning," *International Journal of Performability Engineering*, vol. 18, no. 11, p. 808, Nov. 2022, doi: 10.23940/IJPE.22.11.P6.808816.

[247] A. Plyer, G. Le Besnerais, and F. Champagnat, "Massively parallel Lucas Kanade optical flow for real-time video processing applications," *J Real Time Image Process*, vol. 11, no. 4, pp. 713–730, Apr. 2016, doi: 10.1007/S11554-014-0423-0/FIGURES/16.

[248] "GitHub - Azzam-Alhussain/FPGA-QHAR: Quantized Human Action Recognition on SoC-FPGA edge board." Accessed: Jun. 01, 2024. [Online]. Available: https://github.com/Azzam-Alhussain/FPGA-QHAR

[249] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, pp. 4278–4284, Feb. 2016, doi: 10.1609/aaai.v31i1.11231.

[250] J. O'neill, G. V Steeg, A. Galstyan, V. N. Balasubramanian, and I. Tsang, "Layer-Wise Neural Network Compression via Layer Fusion," *Proceedings of Machine Learning Research*, vol. 157. PMLR, pp. 1381–1396, Nov. 28, 2021. Accessed: Jun. 01, 2024. [Online]. Available: https://proceedings.mlr.press/v157/o-neill21a.html

[251] A. Mishra and D. Marr, "Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Nov. 2017, Accessed: Jun. 06, 2024. [Online]. Available: https://arxiv.org/abs/1711.05852v1

[252] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep Mutual Learning," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4320–4328, Jun. 2017, doi: 10.1109/CVPR.2018.00454.

[253] Y. H. Chen, T. J. Yang, J. S. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE J Emerg Sel Top Circuits Syst*, vol. 9, no. 2, pp. 292–308, Jul. 2018, doi: 10.1109/JETCAS.2019.2910232.

[254] M. Xu *et al.*, "A Survey of Resource-efficient LLM and Multimodal Foundation Models," Jan. 2024, Accessed: Jun. 08, 2024. [Online]. Available: https://arxiv.org/abs/2401.08092v1

[255] J. Kim, Y. Bhalgat, J. Lee, C. Patel, and N. Kwak, "QKD: Quantization-aware Knowledge Distillation," Nov. 2019, Accessed: Jun. 08, 2024. [Online]. Available: https://arxiv.org/abs/1911.12491v1

[256] J. Xu, Y. Pan, X. Pan, S. Hoi, Z. Yi, and Z. Xu, "RegNet: Self-Regulated Network for Image Classification," *IEEE Trans Neural Netw Learn Syst*, vol. 34, no. 11, pp. 9562–9567, Jan. 2021, doi: 10.1109/TNNLS.2022.3158966.

[257] S. Woo *et al.*, "ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders," pp. 16133–16142, Jan. 2023, doi: 10.1109/cvpr52729.2023.01548.

[258] "torch.onnx — PyTorch 2.3 documentation." Accessed: Jun. 10, 2024. [Online]. Available: https://pytorch.org/docs/stable/onnx.html

[259] M. Oquab *et al.*, "DINOv2: Learning Robust Visual Features without Supervision," Apr. 2023, Accessed: Jun. 13, 2024. [Online]. Available: https://arxiv.org/abs/2304.07193v2

[260] A. Kirillov *et al.*, "Segment Anything," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 3992–4003, Apr. 2023, doi: 10.1109/ICCV51070.2023.00371.

[261] H. Cai, J. Li, M. Hu, C. Gan, and S. Han, "EfficientViT: Multi-Scale Linear Attention for High-Resolution Dense Prediction," *ArXiv*, vol. 1, pp. 1–15, May 2022, Accessed: Jun. 13, 2024. [Online]. Available: https://arxiv.org/abs/2205.14756v6

[262] H. Wang *et al.*, "BitNet: Scaling 1-bit Transformers for Large Language Models," Oct. 2023, doi: 10.48550/ARXIV.2310.11453.