UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME DOS SANTOS KOROL

Optimizing for Adaptive CNNs on FPGAs: A Multi-Level Approach

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Antonio Carlos Schneider Beck

Porto Alegre
June 2024

CIP — CATALOGING-IN-PUBLICATION

dos Santos Korol, Guilherme

Optimizing for Adaptive CNNs on FPGAs: A Multi-Level Approach / Guilherme dos Santos Korol. – Porto Alegre: PPGC da UFRGS, 2024.

175 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2024. Advisor: Antonio Carlos Schneider Beck.

1. Adaptive Inference. 2. Deep Neural Networks. 3. FPGA. 4. IoT-Edge Continuum. I. Beck, Antonio Carlos Schneider. II. Título.

AGRADECIMENTOS

Agradeço a todos que colaboraram com a construção dessa tese, nos momentos de dificuldade (atravessando plena pandemia) e nos momentos bons, de comemoração. Agradeço meus pais, Iris e Ricardo, e meu irmão, Leo. Ao Vicente, agradeço nossa amizade e brincadeiras de LEGO para uma pausa da vida adulta. Em especial, agradeço minha esposa, Joice, pelos muitos conselhos ao longo desses quatro anos de doutorado. Tua parceria foi e é fundamental para todas minhas conquistas.

Não posso esquecer também dos meus colegas de laboratório pelas inúmeras ajudas com códigos, escrita de artigos, elaboração de gráficos e afins. Grato também pelos colegas de UFRGS pelas conversas de corredor e cafés pós almoços, ajudando a suavizar o dia-a-dia e tornando a experiência do doutorado mais suportável. Deixo meu agradecimento ao Prof. Caco que durante mestrado e doutorado me orientou com paciência e dedicação.

ABSTRACT

Deep Learning, and especially Deep Neural Networks (DNN), drives the Internet of Things (IoT) revolution. With intelligent components that can interact with other devices, applications ranging from smart cities to healthcare and wearables will permeate our everyday life. To meet latency and privacy requirements in this new application domain, resource-hungry DNN tasks have been migrating to the edge, where IoT devices can offload the inference processing to local edge servers. Processing at the IoT-edge continuum presents an alternative to the far-located cloud, where processing may take more energy, communication time, and data exposure. In this context, novel hardware solutions have also been explored to accelerate specific operations of DNNs, providing significant efficiency gains. For example, FPGAs offer relatively low non-recurring engineering and production costs at good flexibility/programmability and power efficiency levels. Nevertheless, relying exclusively on new hardware devices will not cope with the constant and rapid increase in the DNNs' computational demands, forcing us to look at this efficiency problem from multiple perspectives: from hardware- to algorithmic-level optimizations. A representative use case is the Convolutional Neural Networks (CNNs) used in many IoT applications. At the same time that it is possible to trade the CNN computational load for quality with algorithmic-level optimizations like quantization, pruning, and early-exit, it is also possible to optimize the hardware accelerating the CNN processing with, for instance, High-Level Synthesis or, even, share the load between multiple nodes with offloading. In this thesis, we argue that all axes must be considered together to optimize inference processing as they influence each other and can be exploited to create a unified and comprehensive design space. What is more, given that the IoT-edge is highly heterogeneous, with multiple concurrent applications, variable workloads, and diverse environments, we must take such optimizations dynamically to adapt the inference processing to current requirements. This thesis presents a framework for combining such multi-level optimizations in a dynamic fashion to deliver efficient inference processing across the IoT-edge continuum.

Keywords: Adaptive Inference. Deep Neural Networks. FPGA. IoT-Edge Continuum.

Otimizando para CNNs Adaptativas em FPGAs: Uma Abordagem Multi-Nível

RESUMO

O Aprendizado Profundo, e especialmente as Redes Neurais Profundas (DNN), impulsiona a revolução da Internet das Coisas (IoT). Com componentes inteligentes que interagem com outros dispositivos, aplicações que vão desde cidades inteligentes até saúde e dispositivos *wearable* permeiam nossa vida cotidiana. Para atender aos requisitos de latência e privacidade nesse novo domínio, tarefas de DNN têm migrado para a borda, onde dispositivos IoT podem transferir o processamento da inferência para servidores locais de borda. O processamento no contínuo IoT-borda apresenta uma alternativa ao processamento em nuvem, que pode consumir mais energia, tempo de comunicação e exposição dos dados. Nesse contexto, soluções de *hardware* inovadoras também têm sido exploradas para acelerar as operações das DNNs, proporcionando ganhos significativos de eficiência. Por exemplo, FPGAs oferecem custos não-recorrentes de engenharia relativamente baixos e procesamento com boa flexibilidade/programabilidade e eficiência energética. No entanto, depender exclusivamente de novos dispositivos de *hardware* não será suficiente para lidar com o aumento constante das demandas computacionais, forçando-nos a considerar o problema de múltiplas perspectivas: desde otimizações ao nível do *hardware* até ao algoritmo. Um caso de uso representativo são as Redes Neurais Convolucionais (CNNs) usadas em muitas aplicações IoT. Ao passo que podemos trocar a carga computacional da CNN por qualidade com otimizações ao nível do algoritmo como a quantização, poda e a *early-exit*, também é possível otimizar o *hardware* acelerando a inferência com, por exemplo, Síntese de Alto-Nível ou até mesmo compartilhar a carga entre vários nodos com *offloading*. Nesta tese, argumentamos que todos os eixos devem ser juntamente considerados para otimizar a inferência, pois eles se influenciam mutuamente e podem ser explorados para criar um espaço de projeto unificado e abrangente. Além disso, dado que o contínuo IoT-borda é altamente heterogêneo, com múltiplas aplicações concorrentes, cargas de trabalho variáveis e ambientes diversos, devemos realizar tais otimizações de forma dinâmica para adaptar o processamento aos requisitos atuais. Esta tese apresenta uma ferramenta para combinar tais otimizações em vários níveis e explorá-las dinamicamente para oferecer processamento de inferência eficiente por todo o contínuo IoT-borda.

Palavras-chave: Inferência Adaptativa. Redes Neurais Profundas. FPGA. Contínuo IoT-borda.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AC	Approximate Computing
AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
ANN	Artificial Neural Networks
CLB	Configurable Logic Block
CONV	Convolutional
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DSP	Digital Signal Processing
FC	Fully-Connected
FMAP	Feature Map
FPGA	Field-Programmable Gate Arrays
FPS	Frames Per Second
GPU	Graphic Processing Unit
GTSRB	German Traffic Sign Recognition Dataset
HLS	High-Level Synthesis
IPS	Inferences Per Second
IoT	Internet of Things
LUT	Look-Up Table
MAC	Multiply-And-Accumulate
MVTU	Matrix-Vector-Threshold Unit
ONNX	Open Neural Network Exchange
PE	Processing Element
QAT	Quantization-Aware Training

QoE	Quality of Experience
SDF	Synchronous Dataflow
SoTA	State-of-The-Art
STE	Straight-Through Estimator
SWU	Sliding Window Unit
TPU	Tensor Processing Unit
VLSI	Very Large Scale Integration

CONTENTS

1 INTRODUCTION

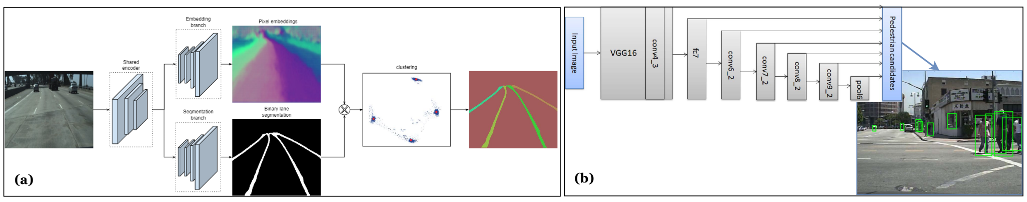
Internet of Things (IoT) is the evolution of internet, with intelligent components, often ubiquitous, with the ability to interact with other devices. The IoT market is expanding consistently (ADEGBIJA et al., 2018). While the global economy showed a 2.3% annual growth in 2022, the electronics production, mostly driven by IoT, grew 4.3% for the same period - producing over 15 Zettabytes of data; and, for the next years, this trend is expect to continue - the IoT is anticipated to collectively produce 38 Zettabytes by 2027 (European Commission for Comm. Networks, Content and Technology, 2023). The growth driven by IoT devices have enabled many and new business across various applications like smart cities, agriculture, healthcare, transportation, wearables, among others (MOHAMMADI et al., 2018b).

Many current and future IoT applications require low latency: the processing output needs to be ready as quickly as possible to provide the response time required for their activity. However, two factors hinder IoT devices to *fully* process the generated data: **Processing Power** - IoT devices are typically composed of simple processors that often have limited processing power, mainly used for data collection and triggering certain controls. As a result, these devices are often unable to process high-volumes of data locally. **Energy Consumption** - Due to their compact and portable nature, many IoT devices rely on batteries. However, battery technology does not progress at the same pace as semiconductor devices and the applications they need to execute. Because of that, there is a trend towards deploying IoT devices that offload the data to be processed elsewhere, at the cloud, and wait for receiving the result back (WANG et al., 2020).

Offloading the computation to the cloud, however, also has its drawbacks. In addition to significantly increasing response time, offloading to the far-located cloud can be energy-intensive and may compromise data security (KANG et al., 2017; CHEN et al., 2021). In this context, the concept of edge computing becomes highly valuable as it allows the data generated by IoT devices to be processed closer to where it is used, rather than sending it to cloud warehouses. Performing such computations at the edge of the network represents a new point in the cost-benefit relationship of energy, latency, security, and resource sharing when compared to cloud processing. The edge offers the benefit of faster responses and higher power efficiency compared to what is provided in the cloud (WANG et al., 2020; NING et al., 2018). Thus, IoT devices (end-nodes) can redirect (all or some of) the collected data to edge servers (sometimes referred to as IoT gateways) that are physically close, creating the IoT-edge continuum.

1.1 Scope and Challenges

Figure 1.1 – Examples of CNN-based applications: (a) encoder-decoder CNNs for road segmentation in a self-driving car application (NEVEN et al., 2018); (b) a CNN for pedestrian detection in an urban monitoring system. (DU et al., 2017).



Be at the edge or at the IoT end-node, the collected data needs to be processed to be of any value to the user. Deep Learning (DL) and its Deep Neural Networks (DNNs), especially Convolutional Neural Networks (CNNs), are essential for various IoT applications for processing text, audio, image, and video (see Figure 1.1 for two examples). At the edge, CNNs can be executed on traditional architectures, such as general-purpose processors. However, these architectures have been proven inefficient as the problem size grows (SZE et al., 2020). The use of Graphic Processing Units (GPUs) has popularized DNNs since users can easily extract parallelism with high-level frameworks like PyTorch (PASZKE et al., 2019) to accelerate training and inference phases (GOODFELLOW; BENGIO; COURVILLE, 2016). However, even GPUs have their limitations: they are highly power and capital-intensive, and despite being generic (programmable), they still have restrictions like handling certain types of data with fixed sizes and requiring processing in large batches.

Therefore, alternative architectural solutions have been explored to accelerate specific operations of CNNs in dedicated hardware, providing significant efficiency gains (SILVANO et al., 2023). For example, Google's TPU (Tensor Processing Unit), which is an Application-Specific Integrated Circuit (ASIC) accelerator for artificial neural networks that is limited to specific types of artificial neural networks and can manipulate only fixed types of data. As the non-recurring engineering and production costs are high to justify the manufacturing of such accelerators, Field-Programmable Gate Arrays (FPGA) came into play as they meet the demand for flexibility/programmability offered by GPUs and the effectiveness provided by ASICs for many applications (BASKIN et al., 2018; NURVITADHI et al., 2017; HAO et al., 2019). Only in the American data center market, the FPGA market size is expected to grow by 25% until 2028, representing more than 25 billion dollars (Global Market Insights, 2023). For FPGA-based deep learning acceleration, we see initiatives both at warehouse scale, like Microsoft's Brainwave (Microsoft, 2023) and Amazon's EC2 F1 Instances (Amazon, 2023), and at the IoT-device level from companies like Aldec (Aldec, 2023), Mipsology (Mipsology, 2023), Synective (Synective, 2023), and many others. However, relying on hardware improvements only may not scale to these modern applications requirements of large models and impressive data volumes (THOMPSON et al., 2021).

Figure 1.2 – Hardware performance versus Deep Learning computing requirements over the years. Adapted from (THOMPSON et al., 2020)

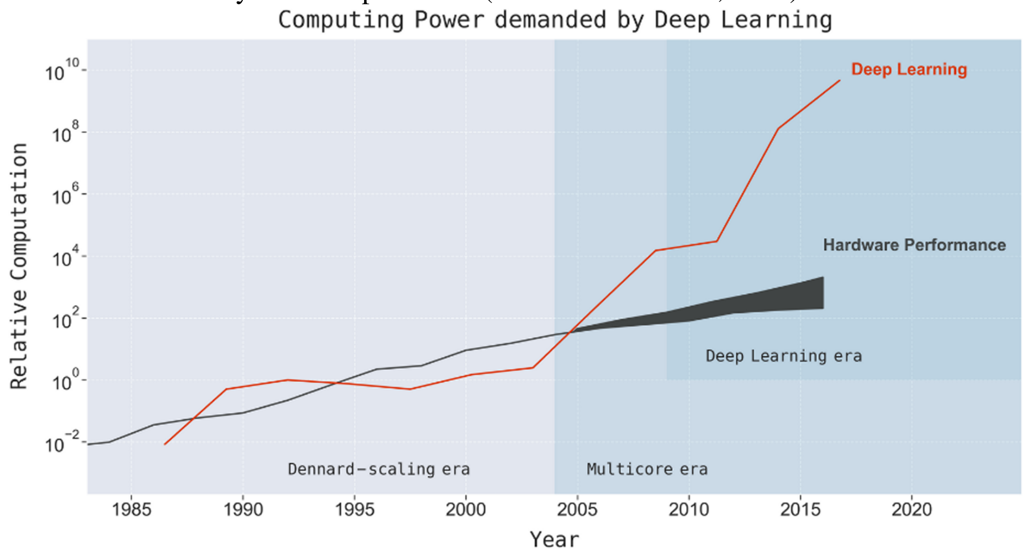


Figure 1.2 demonstrates the gap between modern DL models and the performance delivered by today's machines. Even though from the late eighties up to the early 2000s, the DL models' complexity followed computing performance; it was around 2012 that (even with the migration of DL to GPUs) the growth in models' complexity drifted apart from the performance delivered by our machines (THOMPSON et al., 2021). The start of the "Deep Learning era" (Figure 1.2) marked a turning point in which hardware could no longer provide seamless support to deep learning advancements. Then, it became clear that *technological advancements and hardware improvements alone are not enough to achieve the desired levels of performance and efficiency - requiring that we also look into the models*.

The neural network model (or architecture) defines the connection type (e.g., feed-forward or recurrent), how many layers, the layers type (e.g., convolutional or fully-connected), the layers size (e.g., how many neurons), among many other neural network parameters. By optimizing the model, we can tune parameters like the number of layers and word length to speed up processing. Differently than using a larger accelerator for faster processing, however, when optimizing the model, we trade-off computational load per quality (e.g., accuracy).

1.2 Contributions of this Thesis

Therefore, we argue in this thesis that all aforementioned axes, from the hardware to the CNN models, must be considered and optimized together as they influence each other and can corroborate to an inference processing of improved efficiency and quality for the final user. The ***general contribution*** of this thesis lies in the optimization of CNNs for execution on FPGAs in the IoT-edge continuum, simultaneously addressing the CNN model and its hardware implementation. And, on top of that, we propose exploiting these optimized models and accelerators in a dynamic and adaptive fashion given that modern IoT-edge scenarios are highly heterogeneous, with multiple concurrent applications, variable workloads, and diverse environments (SHUAI et al., 2020; FU et al., 2014; REDDI et al., 2020).

More specifically, the ***first contribution*** focuses on constructing a design space from simultaneous (and complementary) optimizations at following three levels:

- At the hardware-level, speed-ups can be achieved by optimizing the FPGA accelerators running the CNNs. High-Level Synthesis (HLS) (ZHANG et al., 2015a) is a well-known design process that can enable optimizations at the hardware level. With HLS, it is possible to tune the accelerators' HLS code to deliver optimized FPGA-based designs[1];
- At the CNN-level, quantization (COURBARIAUX et al., 2016), pruning (LI et al., 2017), and early-exit (TEERAPITTAYANON; MCDANEL; KUNG, 2016) have been shown to be very prominent alternatives and will be used in this work. Quantization reduces the bit-width across the CNN's weights and intermediate values, saving storage and computation requirements. Pruning removes parts of a CNN to improve performance at the cost of accuracy, while early-exit adds branches to a CNN so that the inference may finish earlier, reducing the processing time;
- At the system-level, the offloading (KANG et al., 2017) optimization can be used to decide *where* the inference is processed in the IoT-edge continuum. The inference

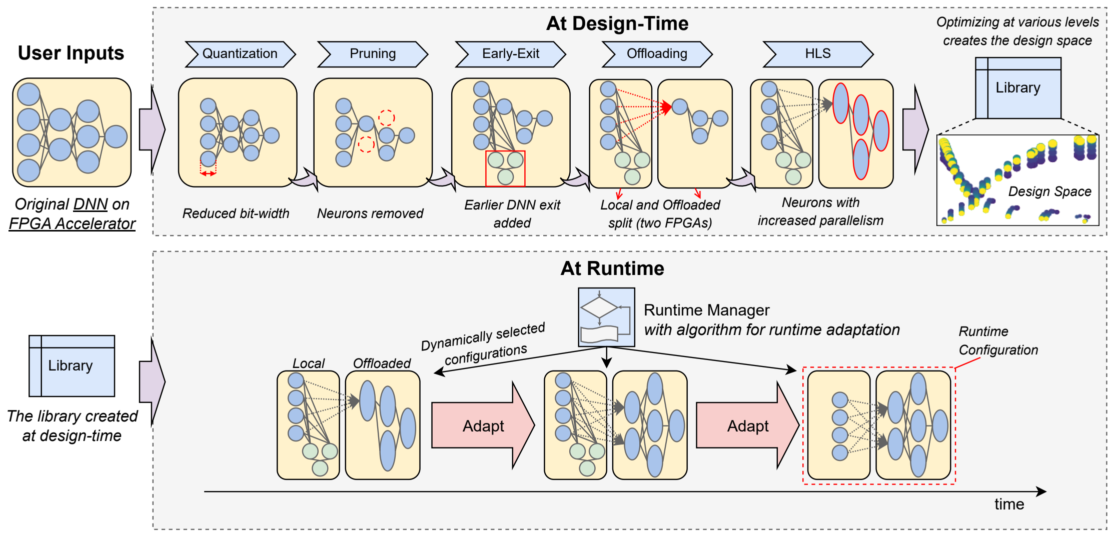
[1]We have also explored approximate FPGA operators for this thesis. However, we opted to not include this optimization in the main text. The work with approximate operators is presented in Appendix D.

processing can be balanced and shared across multiple devices on the continuum, multiplying the design options (e.g., processing only some of the layers locally).

The ***second contribution*** observes the variations in users' behavior, such as the use of multiple applications, increasing data volume, and dynamic IoT environments to design an infrastructure, built on top of the the first contribution, for *adaptive* inference processing that dynamically covers different Pareto points in the design space. Considering an IoT-edge continuum featuring FPGAs at both ends, this contribution dynamically configures the inference processing (i.e., selecting the level of the CNN, FPGA, and offloading optimizations) regarding runtime requirements from user and environment.

Figure 1.3 sketches the general contribution. Our proposal embraces a design-time phase (upper part in Fig. 1.3) and a runtime phase (lower part). Based on the user's DNN and FPGA accelerator input (top left corner), a series of optimizations take place. Quantization, pruning, early-exit, offloading, and HLS are each applied at various levels to create multiple versions for creating a unified *design space*. This design space is saved in the form of a Library (top right corner of Fig. 1.3), feeding the runtime phase (input from bottom left corner of Fig. 1.3). At runtime, the system adapts the inference processing according to some runtime requirements from either user or environment. The adaptation is carried out by the Runtime Manager in charge of changing the *Runtime Configuration* that sets the level of each optimization.

Figure 1.3 – Sketch of the proposed thesis.


At Design-Time
User Inputs
Quantization
Pruning
Early-Exit
Offloading
HLS
Optimizing at various levels creates the design space
Library
Design Space
Original DNN on FPGA Accelerator
Reduced bit-width
Neurons removed
Earlier DNN exit added
Local and Offloaded split (two FPGAs)
Neurons with increased parallelism
At Runtime
Library
The library created at design-time
Runtime Manager with algorithm for runtime adaptation
Dynamically selected configurations
Local
Offloaded
Adapt
Adapt
Runtime Configuration
time

The bottom of Fig. 1.3 illustrates this runtime adaptation with a toy example. In it, three different Runtime Configurations are selected from the Library produced at design-time. Let us consider that, after some time running with the first Runtime Configuration (left-most one), the user increases, for example, the quality requirement. Then, the Runtime Manager searches the library for some other configuration delivering an accuracy higher than the current one. Once found, the configuration is updated (first 'Adapt' arrow). In the example, the Runtime Manager selected a new configuration with more neurons to the right side of the DNN (the layers running after the offloading), increasing the accuracy. After some time, let us say that the FPGA running before the DNN split needs to reduce energy consumption (e.g., due to a battery running low). In that case, a new Runtime Configuration gets selected from the library and the FPGA is updated (i.e., reconfigured) again (second 'Adapt' arrow). This last configuration removes the early exit from the FPGA, which would, in turn, free some of its resources and lower its power dissipation. In the following chapters of this thesis, we detail the role each optimization plays on the generation of the design space, its exploration at runtime to adapt the inference processing, and how this approach evaluates against state-of-the-art solutions.

1.3 Structure of this Thesis

The remainder of this document is organized as follows. Chapter 2 presents the background on Deep Neural Networks, their optimizations, and the FPGA accelerators that are relevant to this work. Chapter 3 presents the literature on DNN optimization with a focus on optimized FPGA-based DNN acceleration and our contributions to the state-of-the-art. After that, Chapter 4 presents the works developed in this thesis. In the end, Chapter 5 gives some limitations of this work and the final remarks.

2 BACKGROUND

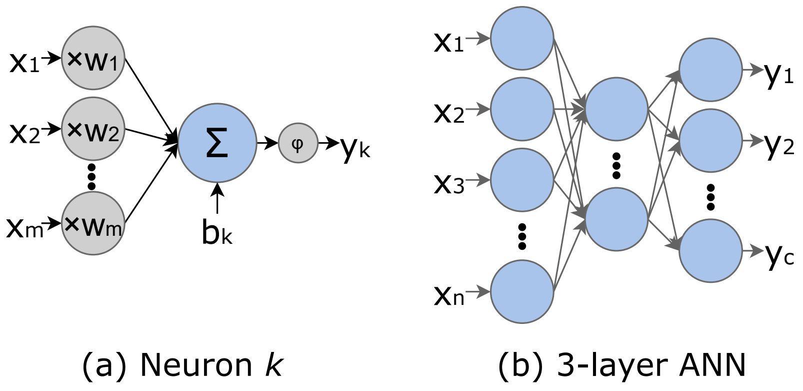
This thesis embraces both algorithm and hardware fields. Thus, this chapter starts covering the theoretical reference on Deep and Convolutional Neural Networks, focusing on their static (Section 2.1.2, including quantization and pruning) and dynamic (Section 2.1.3, including early-exit and offloading) optimizations. The chapter, then, moves to the hardware aspects involved in the CNN execution on reconfigurable platforms (i.e., HLS). In this last part, we detail the two type of accelerators used here, the single-engine (Section 2.2.2.1) and dataflow (Section 2.2.2.2) accelerators.

2.1 Deep Neural Networks

Artificial Intelligence (AI) goal, as put by John McCarthy (MCCARTHY, 2007), is to build machines that are *intelligent* and capable of achieving tasks *like humans do*. One approach to building intelligent machines is to make them *learn* (MITCHELL et al., 1997). By exposing data to a machine, it can learn without being explicitly programmed to do so (SAMUEL, 2000). This approach is called **Machine Learning**.

Given the many possible tasks and applicability, the approach taken to teach these ML algorithms change. In this thesis, we are interested in the Supervised Learning approach since it covers our use cases with Artificial Neural Networks (ANN).

Figure 2.1 – (a) Artificial Neuron and (b) a sample ANN with three layers.



At the core of ANNs is the artificial neuron. Figure 2.1(a) depicts an artificial neuron. A neuron receives several signals (synapses) from other neurons. Each synapse is multiplied by a particular weight. Then, they are all accumulated and added to a *bias*. Finally, the resulting value is passed through an *Activation Function* generating the output synapse. The activation function is non-linear and models the threshold behavior of neurons that causes the neuron to generate (fire) an output only when the combined input signals are greater than a certain value. Another factor is that without a non-linear function, a (sequence of) neuron(s) could be collapsed into a single linear equation, limiting the neuron's representation capability. Below, we express the artificial neuron as an equation for a neuron k with m inputs and activation function φ.

$$y_k = \varphi(b_k + \sum_{i=1}^{m} w_k^i * x_k^i) \tag{2.1}$$

Artificial neurons can be connected so that their synapses propagate from neuron to neuron to increase their collective learning capability. Usually, these connections are structured as *layers* of parallel neurons, where each neuron from a layer l is connected to all neurons in layer $l+1$ (because of that, these layers are also called Fully-Connected). Here, a layer can be expressed as a multiplication between the layer input values (synapses on all of its neurons or x_k^i in Equation 2.1) and the weights of all neurons (w_k^i in Equation 2.1) such as

$$f^l = \varphi(\boldsymbol{x}^T \boldsymbol{w} + \boldsymbol{b}) \tag{2.2}$$

for a layer l with activation φ and tensors $\boldsymbol{x}$, $\boldsymbol{w}$, and $\boldsymbol{b}$ for input, weights, and bias, respectively. See Figure 2.1(b) for an ANN example with three layers. The layer in the far left is the input layer that receives some values, processes them, and propagates to the following layer. This hidden layer feeds the last one, called the output layer. In other words, a three layer ANN f can be viewed as $f = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ for an input x. In this manner, data flows in one direction from the input to the output layer. Such models are called *feedforward* models [1]. Another way to look at an ANN is as a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\Theta})$ from the input $\boldsymbol{x}$ and parameters $\boldsymbol{\Theta}$ (weights and biases) to the output $\boldsymbol{y}$.

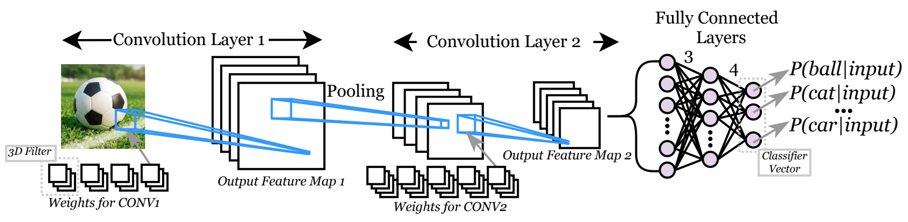
ANNs that have more than one hidden layer (i.e., more than three layers) are called **Deep Neural Networks** (DNNs). This class of networks will be the focus of this thesis. The success behind DNNs is their ability to learn high-level abstract features. The general idea is that with each deeper layer a new set of more high-level features can be represented (and learned) in the model. That property increased the success of DNNs for complex prediction tasks considerably. It made DNNs achieve human-like or even superior performance than humans for specific tasks (e.g., the *ResNet* neural network beating human-level performance in a classification task in 2015 (HE; ZHANG et al.,

[1]ANNs may also have feedback between layers, but those are not in the scope of this thesis.

2016)).

2.1.1 Convolutional Neural Networks

Figure 2.2 – A sample CNN.



One type of DNN that has been broadly used is the **Convolutional Neural Network** (CNN - Figure 2.2). These models have been proposed to deal with grid-like data (e.g., images or time sequences). As their name suggests, CNNs employ the *convolution* operation in some of their layers. Slightly different from the formal mathematical convolution $\int x(a)w(t-a)\,da$, many DNN implementations perform the convolution operation (sometimes represented by an $*$) on discrete and multidimensional operators (e.g., on images) such as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.3}$$

for a two-dimensional input I and two-dimensional kernel K. Usually, for CNNs, we call the input I the input feature map (or input fmap), the resulting matrix S the output feature map (or output fmap), while the kernel K can also be called the convolutional weights or *filter* for a set of kernels that operate on the same input fmap.

The convolution offers interesting properties like *parameter sharing* and *equivariant representations*. Parameter sharing comes from the fact that the convolutional kernels are usually designed to be much smaller than the input fmap. That means that the same small kernel is repeatedly applied over many fmap regions. In contrast with the traditional fully-connected layer (Equation 2.2) where each weight is used only once (multiplied by a particular input value), a weight in a convolutional layer is multiplied against many input values (i.e., reused). This sharing of weights (parameters) grants a smaller memory footprint for convolutional layers and fewer memory loads than fully-connected layers. Equivariant Representation is another exciting property that gives CNNs the ability to their output to respond in the same way that the input changes. That means that, with images, for example, the output feature maps *follows* the feature in the input fmap. For instance, in a convolutional layer extracting edges in an image, if the edge moves from one location to another in the same image, the response in the output fmap will also move by the same amount. It also implies that the kernel in charge of extracting a certain feature (e.g., edges) can be applied over all input fmap.

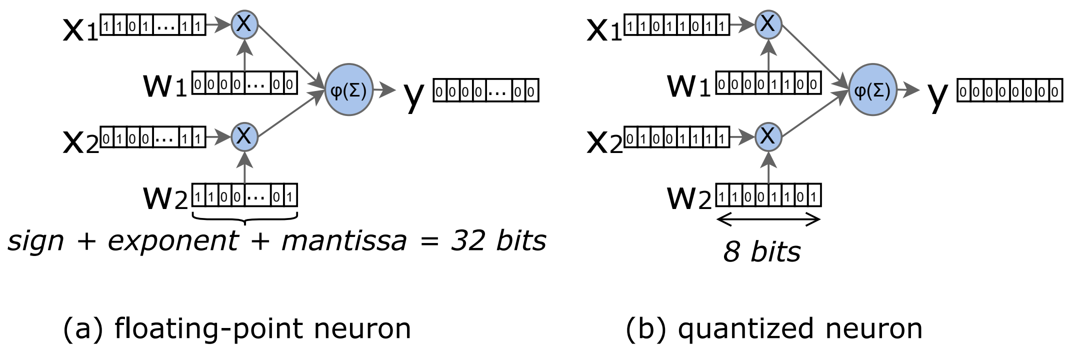
Like the traditional fully-connected (FC) layers, convolutional (CONV) operations are followed by non-linear functions (activations). Some CONV layers may also contain a *pooling* operation (e.g., between convolutional layers one and two in Figure 2.2). Generally speaking, a pooling operation considers smaller regions of the fmap to operate on and generates a *summary* of that region. Some examples of pooling are average-pooling, weighted average, and max-pool. Pooling an output fmap has more than one benefit. First, it reduces the dimensions of the fmap. Second, it adds to the convolution the translation invariance property (GOODFELLOW; BENGIO; COURVILLE, 2016). This property is relevant to extracting input features regardless of their relative location in the input fmap. That can mean either space (e.g., the same feature located in different regions of an image) or time (e.g., the same feature delayed/shifted over input samples).

2.1.2 Static Optimizations

Convolutional and fully-connected layers require extreme amounts of computation and memory transfers. For example, AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) and VGG-16 (SIMONYAN; ZISSERMAN, 2015) CNNs have five and 13 CONV layers, respectively. In both models, three FC layers follow the CONV layers. For a single inference (prediction on one image) on 1000 classes of the ImageNet dataset (RUSSAKOVSKY et al., 2015), CONV and FC layers together constitute 62M weights and 724M multiply and accumulate (MAC) operations in the AlexNet and 138M weights and 15.5G MACs in the VGG-16. Such extreme computational demands have driven research to find more efficient deep-learning models. This section will present the statically applied optimizations covered in this thesis. Namely, quantization and pruning. These optimization methods are traditionally applied before the DNN deployment, usually requiring some kind of re-training or changing in the hardware platform.

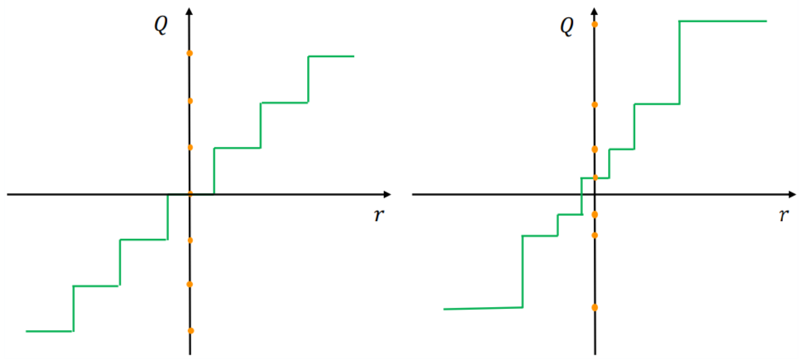
2.1.2.1 Quantization

Figure 2.3 – A 32-bit floating-point neuron (a) and its 8-bit quantized version (b).


X1
W1
X2
W2
φ(Σ)
y
sign + exponent + mantissa = 32 bits
8 bits
(a) floating-point neuron
(b) quantized neuron

Quantization is a method for lowering the cost of a neural network. Quantization for DL reduces the precision of operands in a neural network - possibly both weights and input data. This is usually done by decreasing the number of bits of the supported type, drastically reducing the memory requirement and computational cost. For example, see a two-input neuron quantized from floating point to integer (weights and activation) in Figure 2.3. *All the CNNs evaluated in this thesis are quantized.*

Figure 2.4 – Uniform (left) and non-uniform (right) quantization from real values (r) to quantized (Q) (GHOLAMI et al., 2021).



The *mapping* from a larger to a smaller bit-width can be done in two ways: uniformly or non-uniformly (see Figure 2.4). The uniform quantization is, in essence, a mapping from real to integer values (GHOLAMI et al., 2021). This method is adopted by most of the deep learning quantization frameworks due to its easier implementation. It returns evenly spaced (uniform) steps. These are called quantization levels. In this work, we will focus on uniform quantization. Popularly, the uniform mapping is implemented with the scaling (S) and zero (Z) parameters to convert a real value $r \in [\beta, \alpha]$ to a b-bit wide integer value $Q \in [-2^{b-1}, 2^{b-1}-1]$ with

$$Q(r) = int(r * S) + Z, \tag{2.4}$$

where int can be any 'round to nearest integer' function (real values outside the $[\beta, \alpha]$ range get clipped to the nearest quantized value). To go in the opposite direction, to dequantize a value, one may use $\bar{r} = 1/S(Q(r) - Z)$.

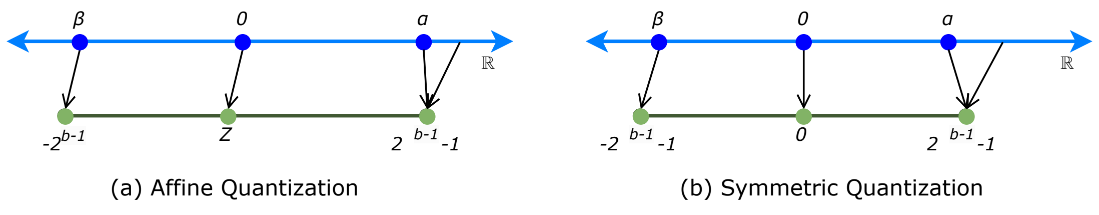
Finding scaling (S) and zero (Z) parameters is crucial, greatly impacting the final DNN performance (JACOB et al., 2018a; KRISHNAMOORTHI, 2018). The first approach to find S and Z is called *affine* or *asymmetric* quantization (GHOLAMI et al., 2021; WU et al., 2020) (see left side of Figure 2.5), where S and Z are defined by:

$$S = \frac{2^b - 1}{\alpha - \beta} \tag{2.5}$$

$$Z = -int(\beta * S) - 2^{b-1}, \tag{2.6}$$

for b-bit quantization. While S divides the range of real values into a number of partitions, Z is rounded to an integer, enabling an exact zero representation. This is important because zero appears in many DNN operations like zero-padding or activation functions like Rectified Linear Unit (ReLU), so not introducing error in the zero representation helps with the final DNN performance.

Figure 2.5 – (a) Affine/asymmetric and (b) scale/symmetric quantization methods.



The second approach is a special case of the affine method. By setting the representable real range to $[-\alpha, \alpha]$, we can set Z to zero. This method is called the *scale* or *symmetric* quantization and simplifies the S and the quantize operation to $S = \frac{2^{b-1}-1}{\alpha}$ and $Q(r) = int(r * S)$, respectively. In this case, the real zero gets mapped to the integer zero, and we end up with a symmetric representation around zero (see right side of Figure 2.5).

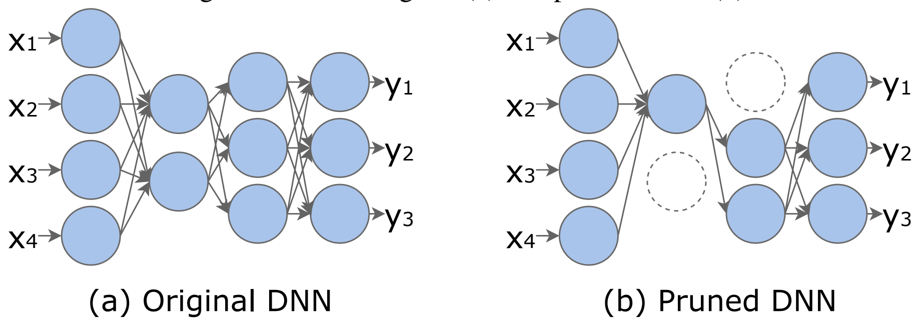
We are left with the definition of the α and β parameters. This can be done by a process called *calibration*, where input samples from the training dataset are used to measure the error and tune the parameters. For instance, it can use the maximum value seen through all the input samples (VANHOUCKE; SENIOR; MAO, 2011). Calibration assumes a pre-trained DNN model, meaning quantization is applied after the complete training phase. These parameters can also be assigned at different granularities, decreasing quantization-caused errors. For instance, we may have a pair of α and β for each layer of the DNN. Defining α and β from a pre-trained DNN may limit the quantization applicability (due to a non-negligible accuracy loss).

A method with increasing popularity to achieve low-impact quantization is Quantization-Aware Training (QAT). QAT trains a DNN while accounting for quantization errors. The rationale behind it is that during the forward pass in a training iteration, the QAT uses the quantized weights and activations (by *simulating* the quantization with Equation 2.4), so the loss calculated at the end already includes the *noise* created by the integer wights/activations. One issue with QAT is that the quantization that now is part of the model has a derivative that is undefined at the step boundaries and zero everywhere else (GHOLAMI et al., 2021). To solve this issue, authors proposed the Straight-Through Estimator (STE) (BENGIO; LÉONARD; COURVILLE, 2013) to replace the original derivative. STE equals 1 for any value in the $[\beta, \alpha]$ range and zero otherwise.

A note on extreme quantization. In the extreme case, we have quantization at the binary (1-bit, also called Binarized Neural Networks, or BNNs) and ternary (2-bit) levels. Naively quantizing to 1-bit integer drastically impacts accuracy (GHOLAMI et al., 2021). Courbariaux et al. showed that the accuracy issue could be circumvented by representing binary weights/activations values with -1 and 1 (*bipolar representation*) and adapting the traditional STE for training (which is one of the reasons behind binary/ternary DNNs being only feasible through QAT) (COURBARIAUX; BENGIO; DAVID, 2015; COURBARIAUX et al., 2016). (LI et al., 2016) and (LIN et al., 2015) added an exact zero representation, creating the ternary DNNs, which improved accuracy at still significant performance gains. Besides the obvious reductions in storage (e.g., from 32-bit floating point to 1-bit representations), binary networks eliminate the need for multipliers. With a binary weights/activations, multiplications can be replaced by XNOR gates followed by bit-counting, which are largely cheaper than either integer or floating point MACs.

2.1.2.2 Pruning

Figure 2.6 – An original (a) and pruned DNN (b).

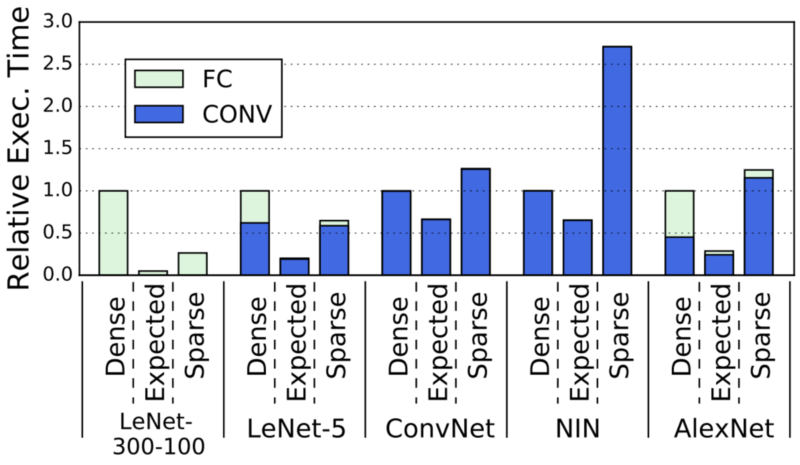


Pruning is a powerful compression technique that explores redundancies in DNNs to reduce the number of weights (for example, see a DNN with two pruned neurons in Figure 2.6). Much of its attention is due to pruned models that achieve accuracy higher than a model of same size trained from scratch (LI et al., 2017). Pruning is especially recommended when there is a need to reduce not only the model's memory footprint (e.g., by quantizing weights), but also operation count (i.e., number of MACs) (CHOUDHARY et al., 2020; MISHRA; GUPTA; DUTTA, 2020). Pruning is also orthogonal to quantization methods, usually tied to the hardware platform.

Denil et al. have shown that modern CNN models with millions of weights are often over-parametrized (i.e., the presence of redundant weights) (DENIL et al., 2013). Those large-size models may help during training, but over-parameterization can cause significant inefficiencies after deployment. However, a direct search for redundant weights is an NP-Hard problem (GUO; YAO; CHEN, 2016) - an ideal pruning eliminates weights at no cost in accuracy.

Pruning can be applied at a finer or coarser granularity. The non-structured pruning methods that remove particular weights or neurons lie at a finer grain. Non-structured methods are known to have a low impact on accuracy. For instance, a representative work by Han et al. (2015) show that it is possible to reduce the weights in the AlexNet CNN from 61 to 6.7 million (a 9× reduction) with no impact on accuracy (HAN et al., 2015). In a larger CNN that offers more 'space' for redundancy, the VGG-16, authors achieved a 13× reduction, from 138 to 10.3 million - again, with no accuracy loss. Han et al. work also set the foundation for iteratively pruning weights. It consists in a three-step approach that first trains the DNN. Then, based on a pre-defined threshold, weights of low importance (i.e., with a value below this threshold) are pruned out. Finally, the pruned model is retrained to recover some accuracy. Whenever the accuracy loss is tolerable (or nonexistent), the model can be pruned and retrained again.

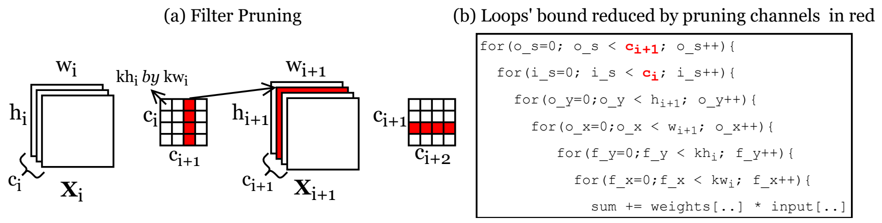
Figure 2.7 – Execution time breakdown of not-pruned (dense) and pruned (sparse) CNNs. The expected execution shows the relative MAC operations in each CNN (YU et al., 2017).



However, the indiscriminate removal of weights from the DNN creates highly sparse weight matrices that may hurt overall performance (YU et al., 2017). Figure 2.7 presents this behavior. Yu et al. compared the execution time of pruned and not-pruned CNN models with the *expected* execution time that is anticipated from the reduction in MAC operations after pruning. As shown in the figure, the expected and sparse bars differ in all evaluated CNNs, showing that pruning may even slow down execution for some models (e.g., ConvNet, NIN, and AlexNet). According to the authors, this is explained by the sparsity created in the weight matrices that are not supported by the traditional memory hierarchy. This issue has motivated the use of pruning at a coarser granularity with the so-called structured techniques. Structured pruning removes whole layers (HE; ZHOU; THIELE, 2018), channels (PENG et al., 2019), or filters (LI et al., 2017).

Filter pruning (LI et al., 2017) is the structured technique used in this work. This technique is applicable to CNNs only and does not create sparsity since filters are removed entirely from the weight matrices of a convolutional layer - see Figure 2.8. This removal also causes a reduction in the number of channels in the output feature map (each channel in an output feature map is created by a specific filter of that same layer). This correlation between pruned filters and their channels grants filter pruning a roughly quadratic effect on reducing the CNN model memory footprint and its respective computations (CHOUDHARY et al., 2020). For example, in VGG-16's first convolutional layer, 64 filters (each has a depth of three channels since they are applied to input images of three channels, RGB) produce an output feature map of 64 channels. Pruning away 50% of this layer's filters means that half of the weights are no longer needed for execution, and the number of channels of the output feature map, which is passed as an input to the next layer, is also reduced by 50%. Moreover, filters of the next convolutional layer can also be reduced since the new input is 32 channels deep.

Figure 2.8 – (a) Filter pruning (LI et al., 2017) and (b) its effect on a straightforward CONV implementation.



```
for(o_s=0; o_s < c_{i+1}; o_s++){
  for(i_s=0; i_s < c_i; i_s++){
    for(o_y=0;o_y < h_{i+1}; o_y++){
      for(o_x=0;o_x < w_{i+1}; o_x++){
        for(f_y=0;f_y < kh_i; f_y++){
          for(f_x=0;f_x < kw_i; f_x++){
            sum += weights[..] * input[..]
```

Formally, we have for a certain convolutional layer i with input feature map $\mathbf{X}_i$ of c_i channels, height h_i, and width w_i, we have c_{i+1} filters F of $c_i \times k_i \times k_i$ (or c_{i+1} kernels of k_i by k_i). After the convolution is performed, it will produce the output feature map $\mathbf{X}_{i+1}$ of $c_{i+1} \times h_{i+1} \times w_{i+1}$. Thus, for a single layer, we have a weight matrix of $c_{i+1} \times c_i \times k_i \times k_i$. By pruning out filters of $c_i \times k_i \times k_i$ from the weight matrix, as explained earlier, we also remove its corresponding channels in the layer's output feature map (red shaded channel over $\mathbf{X}_{i+1}$ in Fig. 2.8(a)). The kernels that compute the removed channels are also deleted in $\mathbf{X}_{i+2}$. Consequently, pruning f filters from layer i saves $f(c_i k_i^2 + c_{i+1} k_{i+1}^2)$ weights and $f h_{i+1} w_{i+1}$ values from feature maps. And, given that the convolutional layer i requires $c_{1+1} c_i k_i^2 h_i w_i$ operations, the pruning of f filters additionally reduces $f(c_i k^2 h_{i+1} w_{i+1} + c_{i+2} k^2 h_{i+2} w_{i+2})$ operations in layer i. For selecting which filters will be selected for pruning, authors in (LI et al., 2017) measure the relative importance of a filter in each layer by calculating the sum of its absolute weight values, which is also known as ℓ_1-norm: $\sum_{j=1}^{n_i} |F_{i,j}|$. After pruning, the model can be retrained to get some of the lost accuracy.

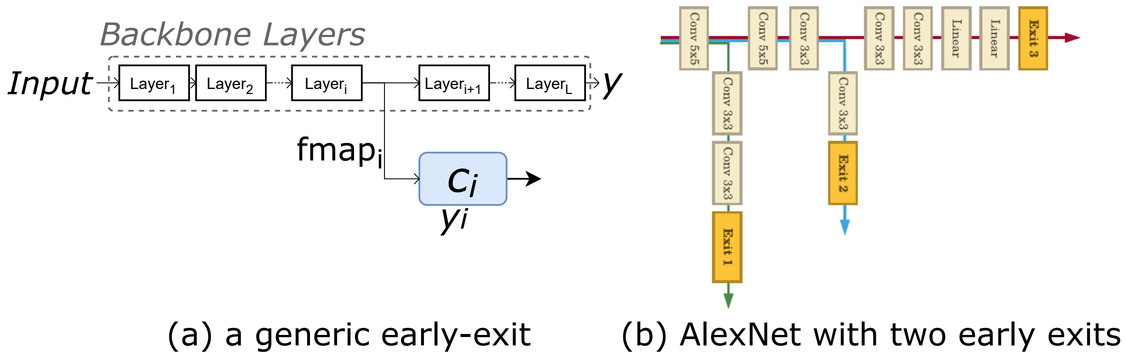
It is important to note that filter pruning has a lower compression capability than most non-structured pruning techniques to the detriment of keeping the CNN hardware-friendly, requiring no modifications to the platform and causing no possible slowdowns due to sparsity. For instance, in (LI et al., 2017), authors were able to compress the VGG-16 and ResNet-110 CNNs in 34 and 38%, respectively, with negligible accuracy drops.

2.1.3 Dynamic Optimizations

The optimizations seen so far can be considered static[2] since they are applied once at design time (before deploying the DNN). Contrary to those, dynamic DNN optimizations allow for some level of *adaptability* of the inference processing. Notable examples of adaptive inference are multi-resolution inference (YANG et al., 2020; HUANG et al., 2017b), dynamic ensembles of DNN models (RUIZ; VERBEEK, 2019), DNNs with dynamic graphs of computations (VEIT; BELONGIE, 2018), and the most popular approaches (LASKARIDIS; KOURIS; LANE, 2021), used here, early-exit DNNs (TEERAPITTAYANON; MCDANEL; KUNG, 2016) and DNNs with split/offloaded inferences (KANG et al., 2017). We opted to use early-exits in this work for two main reasons: it has been shown that quantized early-exits work (KONG; NUNEZ-YANEZ, 2022) and that early-exit is orthogonal to pruning (as shown in Section 4.3). Offloading, on the other hand, was selected since it will enable a wider design space to be explored (when combined with the other optimizations) and leverages the IoT-Edge approach of sharing computing resources across the network.

2.1.3.1 Early-Exit

Figure 2.9 – (a) A generic early-exit CNN and (b) AlexNet with early-exits from (TEERAPITTAYANON; MCDANEL; KUNG, 2016).



Early-exit (PANDA; SENGUPTA; ROY, 2016; TEERAPITTAYANON; MCDANEL; KUNG, 2016) takes advantage of certain inputs being "easier" to process. For these "easy" inputs, not all layers of the DNN are needed, so it can finish earlier (i.e., in a layer

[2]There are works implementing such optimizations in a dynamic fashion. However, those do not represent these optimizations as initially proposed. Those works will be addressed in State-of-The-Art Section.

prior to the last) by following these so-called exits or branches (connected to the DNN original layers, called *backbone*). See an example in Figure 2.9(a) with a generic early-exit CNN. Generally speaking, multiple exits can be added to a DNN and each is able to produce a classification (i.e., output a classification vector) from the same input. The feature map ($fmap_i$ in Figure 2.9(a)) at each branch is sent to the layers of that exit to be classified. Nevertheless, the inference may still continue at the branching point (from $layer_{l+1}$ in Figure 2.9(a)) depending on some decision made based on the y_i exit classification (more details on how this decision is made will follow). In other words, an early-exit DNN will have a set C of exits such that $C \subseteq \{1, ..., L-1\}$ for a DNN with L layers. Therefore, each exit of C takes the i-th output feature map to its classifier c_i, generating output vector y_i of K classes with the probability of the input being a member of each class (i.e., the actual classification) from the softmax function ($\sigma(y)_i = e^{y_i} / \sum_{j=1}^{K} e^{y_j}$).

In this case, an auxiliary classifier c_i is a sequence of layers (convolutional, pooling, fully-connected, etc.). For example, see Figure 2.9(b) for a real-world example of AlexNet with two early exits (*Exit* boxes hold the final fully-connected layers) - note the different configuration (i.e., number of convolutional layers) between the exits. Placing and configuring early exits is an active area of research. Many works have proposed methodologies to design such exits. Kaya et al. have suggested a rule-of-thumb to place exits at every 25% of the DNN MACs (KAYA; HONG; DUMITRAS, 2019a). More recent works have proposed automatic placement of early exits treating it as an optimization problem and solving it with greedy algorithms (PANDA; SENGUPTA; ROY, 2017), or more elaborated solution like Neural Architecture Search (NAS) (TERMRITTHIKUN et al., 2021; ZHAO et al., 2021).

The training takes place after the DNN has been fully defined (i.e., early exits placed and configured). There are two approaches to training early exits (LASKARIDIS; KOURIS; LANE, 2021): exit-only or end-to-end. A set of pre-trained layers (i.e., the original CNN - backbone) is used in exit-only approaches (sometimes called intermediate classifiers only or layer-wise). This approach allows to attach exits to already deploýed DNNs since the exits are trained afterwards. This is achieved by *freezing* the backbone layers so only the weights from exit layers get updated at training. This also means that each exit is trained separately. Since exits are trained one at a time, there is a risk that they 'learn the same classifier' (LASKARIDIS; KOURIS; LANE, 2021). To overcome this issue, works (LEONTIADIS et al., 2021) have employed knowledge distillation (HINTON; VINYALS; DEAN, 2015) between the exits to ensure some differentiation between them to increase accuracy. Despite the advantage of using a pre-trained backbone (original model), this training approach usually returns a limited accuracy (compared to end-to-end training). As we will see next, training all exits (in an end-to-end fashion) improves accuracy and enables greater freedom when designing the early-exit model (LASKARIDIS; KOURIS; LANE, 2021).

End-to-end (or joint) training refers to the methods that consider all exits together during training. Branchynet (TEERAPITTAYANON; MCDANEL; KUNG, 2016) set the ground rules for jointly training early exits and is the approach followed by this work. This method proposes a *joint loss function* that combines all the exits' losses and the backbone loss:

$$L_{joint}(\hat{y}, y, \Theta) = \sum_{n=1}^{N} w_n * L(\hat{y}_{exit_n}, y, \Theta), \tag{2.7}$$

for N exits of weight w and loss L with output $\hat{y}$, ground truth y, and weights Θ. The exits' weights are used to tune the loss function. For example, the Branchynet authors suggest weighting the first exit at 1.0 and the remaining at 0.3.

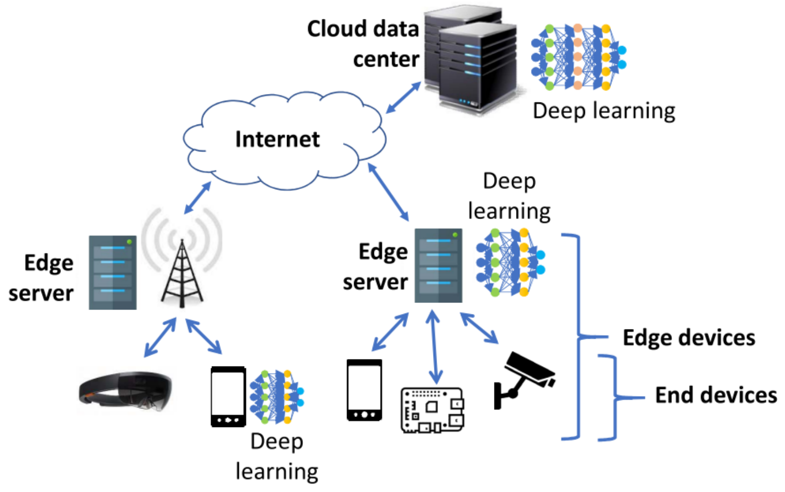
After training, when running inferences, the early-exit CNN must decide whether or not to take earlier exits (e.g., accept the early output in Figure 2.9). This decision is usually based on the confidence that the current input has already been correctly classified with the layers processed so far. When an exit outputs a classification with confidence above a specific value (called *Confidence Threshold*, from 0 to 100%), the output is accepted, and the inference is completed. Using the softmax (the same function used to get the classes probabilities) of the exit output vector is one popular way to measure the exit confidence. So, high probability values mean high confidence. Another way to assess the confidence is by measuring the entropy of the output vector. For example, Algorithm 1 presents the procedure taken by Branchynet for inference (with N exits, confidence threshold T_n, and input x) that uses the entropy of the output vector.

Algorithm 1 *Branchynet inference (TEERAPITTAYANON; MCDANEL; KUNG, 2016)*

```
1: for For $n$ = 1...$N$ do
2:     $z = exit_n(x)$
3:     $\hat{y} = softmax(z)$
4:     $e \leftarrow entropy(\hat{y})$
5:     if $e < T_n$ then
6:         Return arg max $\hat{y}$
7: Return arg max $\hat{y}$
```

2.1.3.2 Offloading

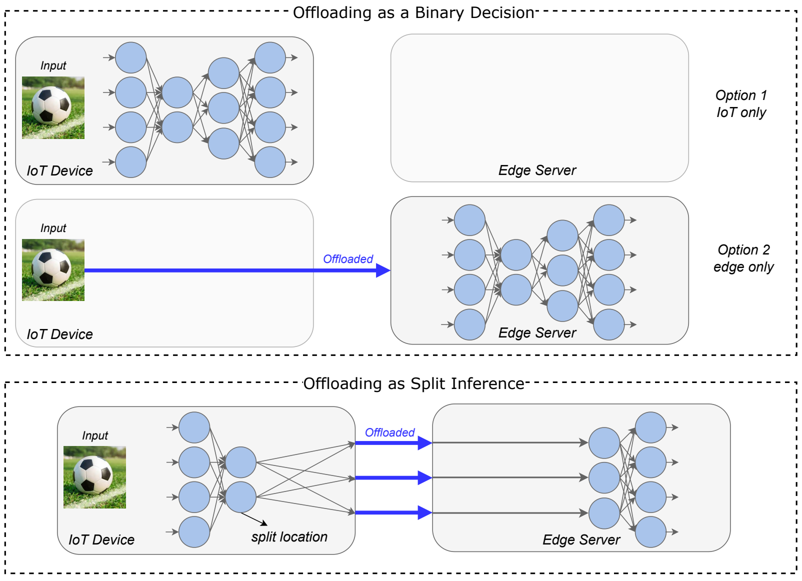
Figure 2.10 – DL models can execute on end-devices, edge, or cloud (CHEN; RAN, 2019).



Inference processing can occur across the stack from end-devices to edge and cloud servers (Figure 2.10). The idea is that devices can *offload* computation, sending their inputs (e.g., images, audio samples) or even intermediate data (e.g., feature maps) over the network from resource-limited devices to servers equipped with high-performance architectures. This thesis focuses on edge computing, where high-performance nodes are placed physically close to the end-devices or sensors. Edge-powered AI is one of the main drivers behind the IoT success (MOHAMMADI et al., 2018a). Compared to cloud-only processing, edge alleviates the network traffic, offers reduced latency, and improves security with less data exposure (WANG et al., 2020; CHEN; RAN, 2019). There are generally two approaches for offloading inferences across the IoT-edge continuum (represented in Figure 2.11).

First, the offloading can be decided in a binary fashion (upper part of Figure 2.11), meaning that the inference is either completely offloaded or it is processed completely local (options 1 and 2, respectively, in the figure). Commonly, this decision is treated as an optimization problem constrained by the network bandwidth/latency, device energy, and/or monetary cost (RAN et al., 2018a; HAN et al., 2016). Offloading computation has been largely studied in the networking literature (like code offloading or Remote Procedure Call), with the difference that now it is also possible to optimize further the computation (i.e., the DNN running inferences), opening up new possibilities (we discuss state-of-the-art works in Section 3).

Figure 2.11 – Two approaches for offloading inferences in the IoT-edge.



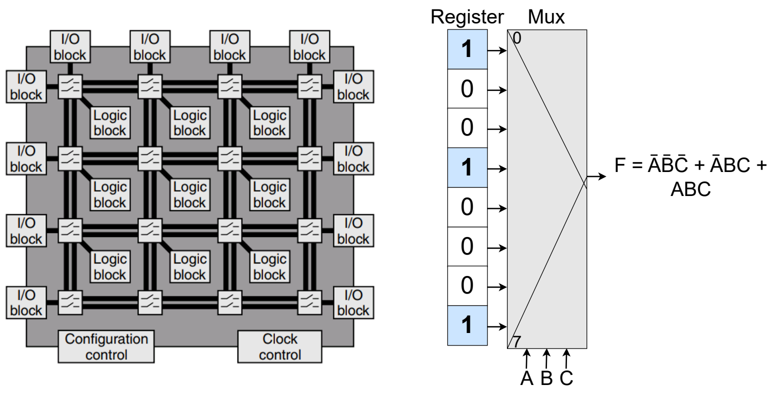
Second, the inference can be partioned (or *split*) when only a fraction of the inference gets offloaded (i.e., sending intermediate feature maps instead of input samples), so only some layers get processed locally. This is depicted in the bottom of Figure 2.11. In such approach, the feature maps produced by the last layer running on the IoT (also known as the *split location*) are sent to the edge server over the network. One of the main advantages of doing so is that the DNN tends to compress the feature maps at each layer, as pointed out by one of the first works on model portioning, Neurosurgeon (KANG et al., 2017). Additionally, the split location (i.e., after which layer to offload) plays a central role in the performance of split inferences. It can be chosen depending on either network conditions, the device computation load, or a combination of both.

2.2 Field Programmable Gate Arrays

The spread of Deep Learning was only possible with the proper hardware support (GOODFELLOW; BENGIO; COURVILLE, 2016). We can even say that deep learning stayed dormant, waiting for a hardware platform that made it feasible to train and execute large models and datasets (HOOKER, 2021). This platform was the Graphic Processing Units (GPUs) that offered enough parallelization and the right memory hierarchy to run the heavy deep learning algorithms on massive datasets. After that, specialized hardware like Google's Tensor Processing Unit (TPU) has been deployed, offering more efficient and faster processing of deep learning workloads (JOUPPI et al., 2017). In this deep learning spectrum between the efficiency of specialized hardware (e.g., TPU) and generic platforms (e.g., GPU), we have the programmable and flexible FPGA. In this work, we focus on the FPGA-based acceleration of deep learning.

Historically, FPGAs have emerged as a cheaper, faster time-to-market, and quicker prototyping platform in contrast to the traditional Very Large Scale Integration (VLSI) systems (BROWN et al., 1992). The key idea behind FPGAs is the *re*-programmability. Its generic logic can be programmed (and re-programmed indefinitely) to implement any function. This full programmability is implemented as a configurable mesh of programmable logic blocks. Configurable input/output blocks usually surround this configurable mesh and contain dedicated clock routing and special blocks like hard-wired multipliers and memory. Figure 2.12(a) shows a generic FPGA organization.

Figure 2.12 – (a) The basic organization of an FPGA (BAILEY, 2011) and (b) A logic schematic of a 3-input LUT with a sample configuration.



(a) Generic FPGA

(b) A 3-input LUT

The programmable logic blocks, the core of the FPGA, are implemented with look-up tables (LUT) in modern FPGAs. LUTs are configurable truth tables. Figure 2.12(b) gives a three-input LUT with an example configuration. For example, Xilinx Ultrascale FPGAs are organized into Configurable Logic Blocks (CLB), where each holds eight six-input LUTs with 64-bit registers and 16 flip-flops (built out of two five-input LUTs) (Xilinx Inc, 2016). Besides CLB and LUTs, modern FPGAs feature memory blocks for internal use (the Block RAM memories, or BRAMs). BRAMs can be inferred by synthesis tools. In most Xilinx families, each BRAM can be configured as two independent 18Kb RAM or a single 36Kb RAM. Moreover, due to its large applicability in Digital Signal Processing (DSP) applications, manufacturers added dedicated ASIC blocks called DSP blocks (or slices) for logic operations and addition/multiplication.

Today FPGAs are also proving useful in other areas, such as high-performance computing, which includes FPGA-based heterogeneous solutions (CHEN et al., 2014; CONSTANTINIDES, 2017). Moreover, Amazon datacenters offer FPGAs for users to run code via *Platform as a Service* (PaaS) (AMAZON, 2023). Microsoft also offers FPGAs in the form of *Software as a Service* (SaaS), primarily used to accelerate applications from Microsoft Azure and Bing (MICROSOFT, 2023).

2.2.1 High-Level Synthesis

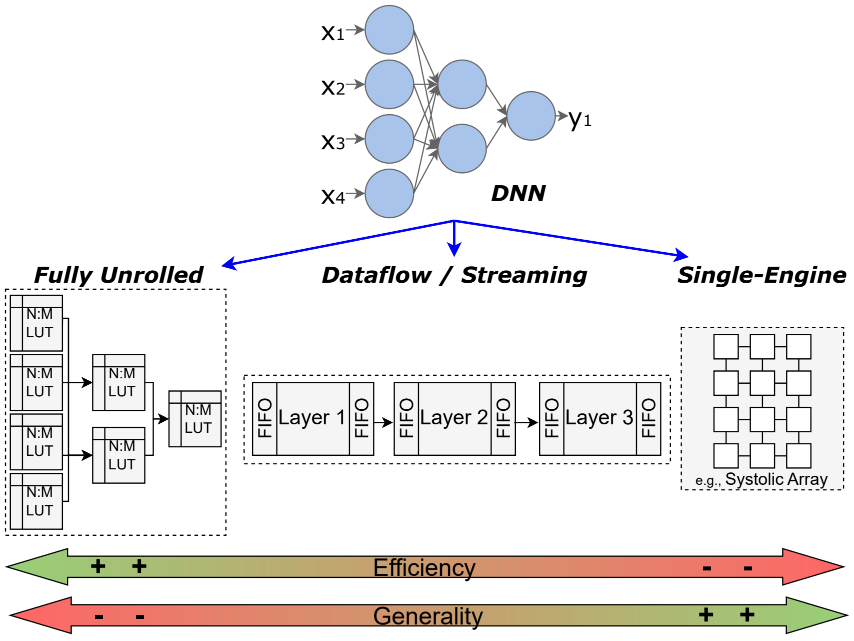
Traditionally, developing an application for FPGA starts with a description in a hardware language (e.g., VHDL or Verilog). This code is synthesized by an FPGA computer-aided design (CAD) tool like Vivado that will perform all the steps up to the configuration of the FPGA device. Such steps can be broken down into synthesis, mapping, placement, routing, and bitstream generation. The bitstream, the equivalent of an executable for software compilation, is the file that configures all logic in the FPGA.

More recently, industry and academia have invested in tools for enabling more abstract descriptions of hardware to ease the development process and speedup the time-to-market. High-Level Synthesis (HLS) allows users to generate a bitstream from a high-level language (e.g., mainly C/C++). Developing hardware through HLS enables users to explore different configurations and tune an FPGA design quickly. The user needs to add pragmas to a, for example, C++ code to guide the HLS tool and configure the size and performance of the hardware to be generated. Many pragmas are available to the user, slightly differing among HLS tools. The most used ones are loop unrolling (to incorporate multiple loop iterations into one), pipelining (to pipeline operations of the loop iteration), dataflow pipelining (coarser level pipelining - e.g., across functions), resource (to specify the FPGA resource to use - e.g., LUT or DSP block), and partitioning (to define the parallelism for data access).

Works by Schafer et al. are among the first ones to treat HLS as a black-box. These works are *supported* by the HLS tool (they are not optimizations *of* the HLS tool). The usual apporach is tuning the use of the synthesis directives for exploiting the design space of FPGA and ASIC deisgins (SCHAFER; TAKENAKA; WAKABAYASHI, 2009; SCHAFER; WAKABAYASHI, 2009; SCHAFER; WAKABAYASHI, 2012). These initial works implemented heuristics (e.g., simulated annealing or divide and conquer) to search the design space. For example, Pham et al. recently proposed a DSE framework that exploits loop-array dependencies to find the best use of HLS optimizations (loop unrolling, pipelining, and array partitioning) for numerical applications (KHANH et al., 2015). The framework proposes a kernel analysis built upon a dependency graph that enables speeding up the exploration time.

2.2.2 DNN Accelerators

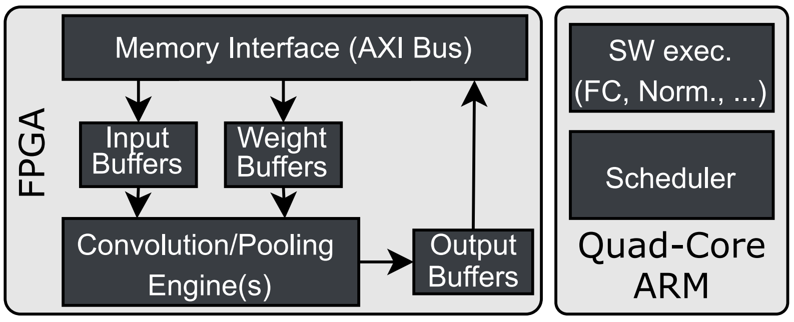
Figure 2.13 – A sample 3-layer DNN and its possible FPGA-based accelerator architectures.



FPGA-based accelerators can be classified according to how they map the DNN to the FPGA fabric (regarding their degree of specialization) - see Figure 2.13. On the one hand, we have fully unrolled accelerators that map every neuron to a particular LUT (or set of LUTs). This mapping represents a one-to-one map between the DNN basic structure (i.e., neuron) and FPGA resources, see left side of Figure 2.13. Because of that, this approach usually returns the highest levels of performance and efficiency (but is limited by the FPGA resources, implementing only relatively small DNNs). At the middle of Figure 2.13, we have the dataflow (or streaming) accelerators at a coarser level. These accelerators leverage the feedforward property of DNNs to map each layer to one hardware module (that are connected in a pipeline fashion through FIFOs). Dataflow accelerators also enable *folding* to reuse FPGA resources to implement the operation in each layer (a dataflow with zero folding is equivalent to a fully-unrolled accelerator). This also allows dataflow accelerators to implement larger DNN models at a reasonable performance. At the most generic end (right side of Figure 2.13), single-engine accelerators execute a DNN in a layer-by-layer fashion. These accelerators employ a single (or just a few) operators (e.g., systolic array or adder tree) to iteratively load and process (a part of) weights and inputs. Besides being able to execute a wide range of DNN models (e.g., without requiring any changes to the FPGA design), they are usually slower than the other two accelerator types. Over the thesis development, we have experimented with the three types of accelerators. However, we opted out from further experiments with fully unrolled accelerators due to its limited scalability. With LogicNets (UMUROGLU et al., 2020) from Xilinx/AMD, the fully-unrolled accelerator evaluated, only small (up to five layers) fully-connected models are supported that do not cope with the more complex ML tasks like image classification used in this thesis. Next, we will detail the two accelerator types covered in this work, the single-engine and the dataflow accelerators.

2.2.2.1 Single-Engine Accelerators

Figure 2.14 – The CHaiDNN-v2 DNN accelerator (Xilinx Inc, 2020).


FPGA
Memory Interface (AXI Bus)
Input Buffers
Weight Buffers
Convolution/Pooling Engine(s)
Output Buffers
SW exec. (FC, Norm., ...)
Scheduler
Quad-Core ARM

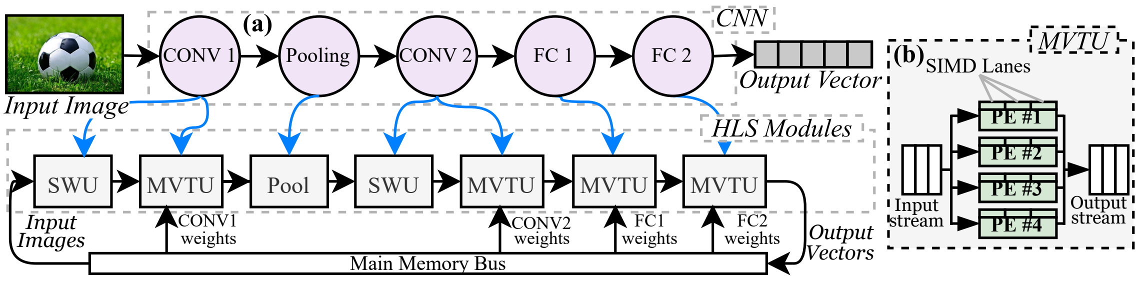
Some popular examples of single-engine accelerators are the Vitis AI, a proprietary IP from Xilinx (Xilinx Inc, 2023), DeepBurning (WANG et al., 2016) from ICT/China, FlexCNN from UCLA/USA (SOHRABIZADEH; WANG; CONG, 2020), DNNWeaver from UC San Diego (SHARMA et al., 2016), and the one used in the first part of this work, the ChaiDNN-V2 also from Xilinx (Xilinx Inc, 2020). CHaiDNN-V2 is an open-source IP that uses convolution engines as basic processing elements (see Figure 2.14). Convolution engines (also configured to perform pooling) are fed by a set of input and weight buffers (implemented as BRAMs). The convolution engines' output is sent to the output buffers. A memory interface is responsible for loading input and weights and storing the output to DRAM through the AXI System Bus, enabling communication between the co-processor and FPGA. With CPU/FPGA collaborative execution, the CHaiDNN-v2 accelerator executes a CNN inference partially on the FPGA (convolutional and pooling layers), and partially on the co-processor. The co-processor is responsible for loading the CNN model, orchestrating data transfers, scheduling, and executing tasks that are not amendable for FPGA, such as image pre-processing (e.g., image resizing and normalization to scale the input image's pixel values to 0-1 range) and the CNN's softmax and fully connected layers.

CHaiDNN-V2 requires DNN models defined in Caffe (JIA et al., 2014), requiring only a post-training quantization (6 or 8 bits fixed-point). CHaiDNN-V2 calibrates the quantization parameters on a subset of training images from the pre-trained model. To that end, the framework offers an auxiliary script called XportDNN that takes the Caffe DNN, the input samples for calibration, and some configuration relating to the numeric format (e.g., CHaiDNN-V2 supports configuring the size of fractional and integer fields, given that they add to up to 6 or 8 bits). At the end of the calibration process, XportDNN outputs a compatible caffemodel and prototxt (the Caffe file formats for storing the DNN topology and weights, respectively) file to be deployed. Then, from the DNN description in the caffemodel, CHaiDNN-V2 constructs a graph of operations (with the corresponding inter-layer dependency) that will appropriately configure the accelerator layer-by-layer.

2.2.2.2 Dataflow Accelerators

Unlike single-engine dataflow accelerators synthesized to a particular DNN model, each dataflow module is configured at design time to implement the layer it was mapped to. The idea behind these accelerators is that they leverage the feedforward nature of DNNs to design a pipeline, achieving throughput usually higher than their single-engine counterparts. There are three main representative frameworks for converting/compiling a DNN to a dataflow accelerator: fpgaConvNet (VENIERIS; BOUGANIS, 2016), HLS4ML (AARRESTAD et al., 2021), and FINN (BLOTT et al., 2018). The two open-sourced, HLS4ML and FINN, were considered in this work. Initial experiments with HLS4ML, however, showed a limitation in the compilation flow with CNNs (i.e., only Multi-Layer Perceptron, MLP, models were supported at the time). Therefore, because we could compile accelerators for state-of-the-art CNN models and have an active open-source repository, we adopted for the FINN framework as our dataflow accelerator.

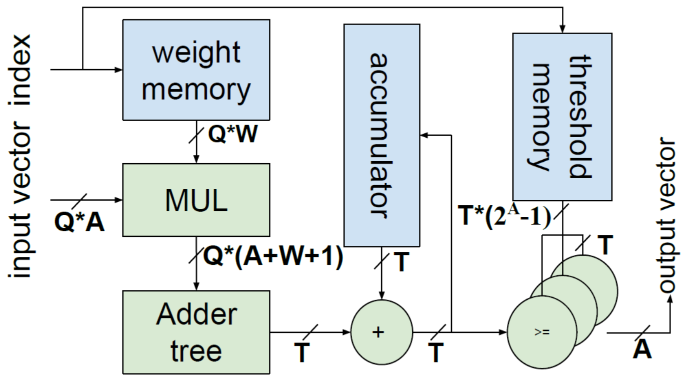
Figure 2.15 – A sample CNN-dataflow mapping.



FINN is a tool from AMD/Xilinx that is being heavily used in academia and industry. FINN employs hardware modules implemented as a set of C++ High-Level Synthesis (HLS) template classes with parameterizable kernel size, stride, etc. The main HLS module in the FINN infrastructure is the Matrix-Vector-Threshold Unit (MVTU) that is used to map CONV and fully-connected (FC) layers - Figure 2.15(b). FC layers get mapped directly to MVTU modules. CONV layers, on the other hand, need an auxiliary module, the Sliding Window Unit (SWU), that prepares the input feature map before it can be multiplied with the weight matrix at the MVTU (*IM2COL* (CHETLUR et al., 2014)). FINN allows the user to tune the accelerator parallelism through a JSON configuration file specifying the number of processing elements (PEs) and SIMD lanes of every MVTU - this is the *Folding Configuration File* (see a commented folding example in Appendix H). Defining PE/SIMD values gives the *folding* of that layer, which directly affects MVTU's performance and resource usage (see the detailed PE in Figure 2.16). We also note that, per default, the FINN accelerator does note use the FPGA's DSP slices, mapping all operations to LUTs.

Figure 2.15(a) shows a mapping from a sample CNN to a simplified FINN dataflow. Note the convolutional layers mapped to SWU + MVTU modules. The mapping from a DNN to an FPGA design happens inside the FINN compiler in ten steps (or transformations) covering exporting a DNN model to preparing it and building the hardware. FINN relies on a Pytorch library called Brevitas (PAPPALARDO, 2023) to train quantized DNNs that can be exported to as an Open Neural Network Exchange (ONNX) (ONNX,

Figure 2.16 – FINN's PE (A and W are the activation and weight bit-width, Q is the SIMD width, and T is the width prior quantization) (BLOTT et al., 2018). Note that MUL and Adder Tree are replaced by XNOR and POPCOUNT for 1-bit quantization.



2023) file. An ONNX file stores the DNN as a graph where each node represents an operation in the network (i.e., a layer). When exported through Brevitas, the ONNX file contains only FINN-friendly operators and can be used to start the compilation process.

FINN starts with the preparation **Tidy-Up** and **Streamlining** steps. They will first name all nodes and data (tensors) in the graph, fold constants, and infer data types. Later, the streamline (UMUROGLU; JAHRE, 2017) transformation rearranges operations in the graph to facilitate (and improve) the quantization. It also includes the *MultiThreshold* operators that implement the quantization and later will be collapsed to the network operations. This 'collapsing' is based on the fact that, much like the MAC in convolutions, the quantization $Q(r) = int(rS) + Z$ (see Section 2.1.2.1) is just another linear transformation. Based on that observation, Umuroglu et al. develop a three-step process that first quantizes weights/activations as successive thresholds (MultiThreshold operator), moves operations closer and collapses linear transformations together, and then absorbs the linear transformation into a threshold. The resulting graph contains fewer MACs and fewer potential quantization-due errors.

Now, FINN can **covert to HLS layers** every node in the ONNX. This transformation will replace nodes with a call to an HLS function from the FINN library (here, it is also possible to configure the function C++ parameters like the number of input and output channels and filter size). Any node that was not converted to HLS will get separated in the following step. The **Dataflow Partitioning** creates an FPGA partition that will follow the synthesis path, leaving out any CPU-like nodes (e.g., input pre-processing and output reading). Next, we have the **Folding Adjustment** to configure the parallelism (i.e., PE and SIMD values) in the HLS modules. FINN can do this configuration automatically when setting a target throughput or manually with the JSON configuration file. This step prepares the ONNX to enter the hardware-building transformations.

The next step is the **HLS IP Generation**. It runs over all modules, synthesizing them and creating their respective Vivado IPs. Then, FINN calls **Create Stitched IP** to connect all IPs in a single TOP module. Some final steps, **Create a Vivado project**, **Synthesize, Place, and Route** to generate the bitstream, **Generate driver** for the co-processor software, and, finally, the **Run on Hardware**. Alternatively, the user can follow Simulation and Emulation Flows with the stiched IP. In this case, FINN calls **Prepare CPP Sim** to simulate a network of HLS layer wrappers or the **Prepare RTL Sim**, where the full accelerator is compiled with Verilator (Veripool, 2021) so an **RTL Simulation** can take place.

3 STATE-OF-THE-ART

This section overviews the works related to this thesis. We start with optimizations at the CNN level (Section 3.1 with quantization, pruning, early-exit, and offloading), and move to hardware-level optimizations (Section 3.2 with high-level synthesis). The chapter ends with a summary of our contributions to the state-of-the-art (SoTA).

3.1 Optimizations at the CNN level

3.1.1 Quantization

Works have mainly quantized DNNs to enable their execution on resource-constrained devices like microcontrollers and to take advantage of the available hardware like FPGAs. In this section we explore the state-of-the-art on quantization, starting with (UMUROGLU; JAHRE, 2017) that presented a method to map DNNs to quantized DNNs (QNN) used in the Brevitas/FINN design flow. We, then, move to works improving quantization as an optimization method to deploy more efficient inference processing systems.

Umuroglu and Jahre developed the *streaming* flow to convert all operations in a quantized DNN to integer operations and proposed a set of bit-serial techniques to run these models on CPU with bitwise operations only (UMUROGLU; JAHRE, 2017). Their work proved later to be crucial to the Brevitas/FINN design flow. Moving all operations to integer enabled the full mapping of DNNs to FPGAs. Some works like XNOR-NET (RASTEGARI et al., 2016) and Binarynet (HUBARA et al., 2016) prior Umuroglu and Jahre used to implement operations like batch normalization and α-scaling on floating-point units to improve accuracy. The authors approach is applicable to any DNN quantized with uniform quantization. It follows three steps. First, “quantization as successive thresholding” that defines a series of thresholds to map a real-valued x to a n-long integer:

$$T(x,t) = \begin{cases} 0, \text{for } x \leq t_0, \\ 1, \text{for } t_0 \leq x \leq t_1, \\ ... \\ n-1, \text{for } t_{n-2} \leq x \leq t_{n-1}, \\ n, \text{for } t_{n-1} \leq x. \end{cases}$$

The second step is the “moving and collapsing linear transformations,” where the authors take advantage of the fact that any sequence of linear transformations can be collapsed together. For example, the same $ax + b$ operation for quantizing activations and performing batch normalization (i.e., at inference, multiplying the input by S or γ and adding by Z or β for quantization (COURBARIAUX; BENGIO; DAVID, 2014) or batch normalization (IOFFE; SZEGEDY, 2015), respectively). Next, the transformations that remained can be “absorbed into thresholds.” The a and b parameters left get mapped to thresholds following $t_i \leftarrow (t_i - b)/a$. The resulting DNN is implemented as a series of matrix multiplication and thresholding pairs.

With the model “streamlined,” Umuroglu and Jahre developed a CPU-based implementation to leverage the integer-only operations. Tailored for few-bits quantization (e.g., 1 or 2), they developed the BinaryGEMM (a library for multiplication of binarized matrices). This library is heavily-based on CPU instructions for bit-serial and logic operations. Recall that for 1-bit quantization, the multiplications can be implemented as a XNOR and POPCOUNT (COURBARIAUX et al., 2016). The general idea is to avoid padding with zeros the not used bits in the CPU datapath (e.g., CPU with 8-bit instructions and a DNN with 1-bit weights/activations). The authors show that these optimizations achieve a 3.5$\times$ speedup over an optimized GEMM library on ARM processors.

Capotondi et al. proposed the CMix-NN for mixed precision DNNs targeting microcontrollers (CAPOTONDI et al., 2020). This framework enables any combination of 8, 4, and 2 bits on each layer weights, inputs, and activations that can be implemented with different quantization strategies. CMix-NN employs uniform quantization (see Section 2.1.2.1), allowing to calculate the scale and zero parameters in per-layer or per-channel fashion. On top of that, the authors provide two compression rules. The first one is called FB that folds the batch normalization weights into the convolution weights before quantization (JACOB et al., 2018b). The second approach is called Integer-Channel Normalization (ICN). It combines all non-convolutional parameters into the activation function (RUSCI; CAPOTONDI; BENINI, 2020). More importantly, CMix-NN enables the user to extensively explore the quantization design space. When considering the same 2MB storage budget, the CMix-NN designs at the latency-accuracy Pareto curve improve accuracy by up to 8% and 23% over the state-of-the-art 8-bit (STMicroelectronics, 2019) and floating-point (LAI; SUDA; CHANDRA, 2018) Mobilenets, respectively.

3.1.1.1 Quantizing CNNs for FPGA

Amiri et al. proposed a multi/mixed-precision DNNs for FPGA (AMIRI et al., 2018). The idea is that to alleviate some of the accuracy cost due to aggressive quantization (i.e., binary weights/activations), a floating-point version of the same DNN is concatenated to the quantized layers. To decide whether or not the floating-point layers are needed, the authors trained a one-layer fully-connected DNN, called Decision-making unit (DMU). This DMU takes the output vector from the binirized layers and outputs a yes/no value to decide to continue the inference to the floating-point layers. The dataset to train the DMU was built with CIFAR-10 by paring the 10-value vectors (input samples) with the target value indicating if that sample had been correctly classified by the binary layers. In other words, the DMU was set to learn where the binary DNN made mistakes. The proposal was evaluated on an FPGA board, where the binary layers are deployed on a FINN accelerator and the floating-point ones on the ARM co-processor. The authors report that the accuracy, when compared to a binary-only inference, increased from 78.5% to 82.5%. When compared to the CPU-only inference, their multi-precision implementation achieved 3.06$\times$ speedup.

Neda et al. (2022) propose a multi-precision CNN accelerator. The proposal lies in using a modular multiplier that can be dynamically configurable to different bit-widths. In the case study explored, authors evaluate a 16-bit by 16-bit multiplier composed of four 4-bit by 4-bit multipliers. Thus, the multiplications can be carried at 4, 8, or 16 bit-widths. These multipliers are used in a systolic array accelerator (single-engine) to accelerate AlexNet and LeNet CNNs, among other MLP models. Authors report that going from 26 to 4-bit multiplications on a Zynq7Z020 FPGA represents a speed-up of up to 15.6$\times$ for MLP models and 12.8$\times$ for CNNs at 3 and 4.7% drop in TOP-1 accuracy, respectively.

Coelho et al. have combined a range of quantization methods in a single library and implemented a method for automatically finding the optimal DNN quantization (Coelho Jr et al., 2021). The work was deployed as the QKeras and AutoQKeras frameworks. QKeras is a quantization-aware framework (similar to Brevitas, presented in Section 2.2.2.2) that extends a Pytorch-based DL framework, Keras, with quantized drop-in replacements for DL operators like convolutional and linear (FC) layers. Once a DNN has been defined with QKeras, it can be trained and optimized by AutoQKeras. Additionally, the proposed QKeras has support for the FPGA dataflow accelerator HLS4ML (AARRESTAD et al., 2021), allowing easy FPGA acceleration. QKeras, like Brevitas, uses STE (BENGIO; LÉONARD; COURVILLE, 2013) to train the quantized models. Within the framework, the different *quantizers* can be used by simply passing them to the layer operator. Seventeen quantizers are supported, from binary and ternary to stochastic and power-of-two.

To navigate this large design space, AutoQKeras, besides the bit-width in a per-layer fashion, finds the optimal number of neurons or filters per layer. In the context of AutoQKeras, the authors proposed the "forgiving factor" that defines a maximum drop in any metric (i.e., accuracy, as caused by quantization and neuron/filters removal) versus the gain in some other metric (energy or model size). We note that this approach is quite similar to pruning. However, Coelho et al. approach this removal of neurons/filters *before* training, meaning that the model is trained from scratch - different from pruning that starts from a trained model when removing parts of it. AutoQKeras supports the trading-off the model's energy or bit-size per accuracy. Formally, the forgiving factor is defined as: $FF = 1 + \Delta_{acc} \times log_R(S \times \frac{C_{ref}}{C_{trial}})$, where Δ_{acc} is the maximum tolerated accuracy loss, R and S are constants defined to tune the FF behavior, and the C_{ref} and C_{trial} are the original and optimized costs (energy or model size). Then, the quantization and number of filters/neurons are included in the Keras Tuner[1] and searched as a hyper parameter with random search, hyperband, or Gaussian processes (a energy model for the layers were also added to Keras). Evaluated on the LHC jet classification dataset, Coelho et al. report that the optimized models achieved accuracy and resource-efficiency higher than the state-of-the-art (LogicNets (UMUROGLU et al., 2020) from Xilinx)

Table 3.1 summarizes the works presented in this section with their main goals and the quantization method proposed. We note that works target different objectives, from reduction in the FPGA resource usage (Coelho Jr et al., 2021) to performance improvements and increases in performance - when given a memory envelope (CAPOTONDI et al., 2020).

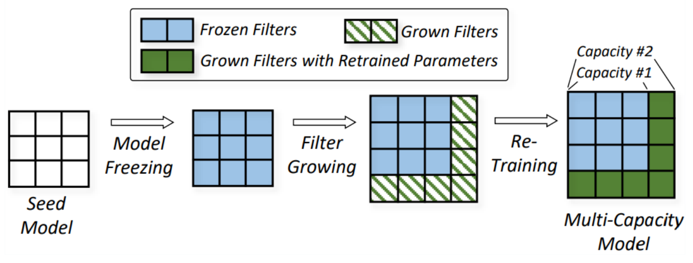
Table 3.1 – Comparison among the selected state-of-the-art works on quantization. WxAy indicates x- and y-bit quantization for weights and activations, respectively. $\uparrow$ for improvement and $\downarrow$ for reduction on speed-up (SU), accuracy (AC), throughput (TH), FPGA resources (RE).

Work	Platform	Baseline	Main Result	Quantization
Umuroglu and Jahre (2017)	CPU (ARM Cortex-A57)	INT8	$\uparrow$ 3.5$\times$ SU	W2A2
Courbariaux et al. (2016)	GPU (GTX750 Nvidia)	W1A1	$\uparrow$ 7$\times$ SU	W1A1 (optimized GPU kernel)
Capotondi et al. (2020)	CPU (ARM Cortex-M7)	FP32 Mobilenet	$\uparrow$ 23.02% AC	Mixed (models under 2MB)
Amiri et al. (2018)	FPGA/CPU (Zynq-7000)	FP32 CPU	$\uparrow$ 3$\times$ TH	W1A1 (+ FP32 on CPU)
Coelho Jr et al. (2021)	FPGA (Xilinx VU9P)	Mixed	$\downarrow$ 50$\times$ RE	Mixed (models under 10ns latency)

[1]https://github.com/keras-team/keras-tuner

3.1.2 Pruning

Figure 3.1 – Freeze-&-Grow for creating multi-capacity CNNs (FANG; ZENG; ZHANG, 2018).



Besides the original works on pruning presented in the Background Section 2.1, here we present the SoTA works on this optimization. We note that even though pruning was originally proposed as a static optimization, it was also extended to be carried out dynamically in some of the SoTA works presented next. The pruning-created resource-accuracy trade-off was exploited by on-device frameworks (FANG; ZENG; ZHANG, 2018; KANG; KIM; PARK, 2019; XU et al., 2019) for mobile vision inference. For instance, NestDNN aims to enable multi-tenant smartphone applications (FANG; ZENG; ZHANG, 2018). For that, NestDNN covers both offline and online stages. At the offline stage, it generates a multi-capacity DNN with a filter pruning technique and builds a number of model versions. Then, these pruned models are fed to a technique called 'Model Recovery' that takes the smallest pruned model, freezes its weights, adds a certain number of filters back, retrains the newest larger model, and iteratively repeats the procedure, building a set of nested descendant versions of a single CNN (a procedure called Freeze-&-Grow, see Figure 3.1). At the online phase, a model profiling measures latency, and memory footprint of all descendant versions. Based on this profiling, the NestDNN scheduler acts when the system's available resources change. When that happens, it reevaluates the selected descendant DNN by jointly optimizing for accuracy and latency. Evaluated on a set of datasets and CNN models against a resource-agnostic baseline, NestDNN increases as much as 4.2% accuracy with 1.7× energy reduction.

ReForm (XU et al., 2019) is another framework that provides a resource-aware inference mechanism. It proposes two optimization schemes. The first one is static that, based on a default resource configuration (i.e., device's memory, energy, and computation capacity), prunes and fine-tunes a pre-trained CNN before deployment. With the dynamic scheme, a model selection for reading the current resource budget. The CNN computational cost is, then, compared to this budget. ReForm will iteratively mask a percentage of the CNN filters until the CNN cost gets below the budget. Across static and dynamic schemes, authors report 16$\%$ latency reduction, 48$\%$ decrease in memory requirement, and 21$\%$ energy reduction.

Dynamic-OFA (LOU et al., 2021) proposes a dynamic CNN that, based on an original model, can switch at runtime between sub-networks depending on a runtime profiler. The sub-network generation works by *sampling* layers from the backbone. So, each sub-network employs only a subset of the backbone layers to create smaller versions, creating different accuracy-latency combinations (i.e., pruning). While the backbone weights (and layers) are shared among all sub-networks, the batch-normalization parameters are re-calculated and stored to recover accuracy. Then, at runtime, the Dynamic-OFA monitors the inference latency and, whenever this value violates a pre-defined constraint, Dynamic-OFA can switch to a smaller sub-network. Evaluated over a state-of-the-art CNN, authors report speedups of up to 3.5$\times$ and 2.4$\times$ over CPU and GPU, respectively.

DiReCtX (XU et al., 2020) proposes another filter pruning-based method for adapting the inference processing to runtime constraints. To prune the CNN in this work, the authors combine the ranking of filters importance (evaluated as the $l1$-norm - the same as the method presented in Section 2.1.2.2) with the each filter's impact on memory, energy, and time consumption. This combination leads to the so-called pruning priority PK as $PK_k = \frac{norm(\alpha_k)}{\beta_e norm(E_k) + \beta_m norm(M_k) + \beta_t norm(T_k)}$, for a filter k with normalized $l1$-norm α_k and energy E_k, memory M_k, and time T_k costs (each weighted by its parameter β). After DiReCtX has calculated the priority of all filters they can be grouped to be pruned out. Evaluated with LeNet, CaffeNet, and VGG-13 CNNs, DiReCtX accelerate inference at up to 44$\%$, reducing energy and memory consumption in up to 32$\%$.

Hawks et al. studied the correlation of quantization and pruning for low-latency inferences in the high energy LHC experiment with MLP DNNs (HAWKS et al., 2021). The authors explore this correlation at training time with the so-called "quantization-aware pruning" (QAP). The motivation behind this study was to integrate the pruning to the training of QNNs. To that end, the authors proposed two integration methods. The first one starts from a trained model, prunes a fraction of the weights, and retrains the pruned model (i.e., by masking the pruned weights). This process can be repeated many times - by monitoring a the accuracy loss against a maximum allowed, for example. The authors call this approach fine-tuning (FT) pruning. The other method is called LT because it is based on the Lottery Ticket hypothesis (FRANKLE; CARBIN, 2018). The LT hypothesis states that "A randomly-initialized, dense neural network contains a sub-network that is initialized such that -when trained in isolation- can match the test accuracy of the original network after training for at most the same number of iterations.” In practice, Hawks et al. follow this LT procedure that 1) initializes the model weights with randomly generated values; 2) trains the DNN for some number of epochs; 3) prunes a percentage of weights; and, 4) *reset* the remaining (not-pruned) weights to their initialized values (from step 1). These steps can also be performed iteratively, returning smaller models at each time. The authors exploit these pruning methods with quantization-aware training in a number of experiments (including the effect of batch normalization and L_1 regularization). Overall, the authors show that, at nearly no cost in accuracy, QAP can reduce the number of operations in 25, 3.3, and 2.2$\times$ over 32-bit floating-point implementation, pruning with post-training quantization, and quantization-only (i.e., QAT), respectively.

3.1.2.1 Pruning CNNs for FPGA

On FPGAs, Faraone et al. (FARAONE et al., 2018) propose a toolflow for statically customizing CNNs for the underlying dataflow accelerator. Their toolflow for hardware-aware pruning follows a heuristic that (i) prunes filters (at 10% steps) while the accuracy stays below a pre-defined threshold; (ii) increases folding[2] while the design exceeds the FPGA resources; (iii) finetunes the CNN to ensure dataflow compatibility (adding or removing filters so no PE in the dataflow is left idle); and, (iv) checks the accuracy a last time to ensure the finetuned CNN kept accuracy above the minimum. Evaluated on AlexNet and TinyYolo CNNs, the toolflow produced designs with throughput and resource usage roughly 2$\times$ better than their original counterparts.

In (YOU; WU, 2020), the authors propose a hardware/software approach to implement sparse convolutions on FPGA. This sparsity is created by an unstructured pruning method (different from the one used in this thesis that does not create sparsity and, thus, does not require a new hardware implementation). The accelerator called RSNN uses a two level architecture that is split into Processing Units (PU) and Elements (PE). The idea lies in creating sparsity (during pruning) matching the width of the PUs and PEs. So, there are no zero weights loaded that would leave MACs idle. Particularly a row of non-zero (i.e., not pruned weights) is loaded into a row of PEs. This PE rows form a PU. One PU feeds the next in a pipeline fashion. After a series of pruning and retraining iterations, the authors produce a CNN with sparsity matching the hardware architecture. AlexNet and VGG-16 are evaluated on a Xilinx Zynq ZC706 FPGA board. When pruning 62.9%

[2]see background section on dataflow accelerators in Section 2.2.2.2.

and 75.0% of the convolutional weights in the AlexNet and VGG-16 the proposed sparse pruning loses 2.2% and 0.62% of the CNNs accuracy, respectively. When compared to other sparse convolution accelerators, RSNN achieves speedups of up to 2.93×.

Véstias designed a pruning method tailored for FPGA acceleration, the so-called configurable block pruning (VÉSTIAS, 2021). The FPGA accelerator is a single-engine based on a systolic array of PEs. Each PE can be configured at design-time to perform 1, 2, 4, or 8 operations in parallel. To increase the accelerator utilization, this block size is taken into consideration during pruning to group weights. The pruning first takes the average of each group weights and ranks them. Then, the groups with the lowest averages are pruned away. Besides pruning, the work proposes a final fine-tuning step that allows to reduce the bit-width from 8 to 4 bits (if the accuracy cost is tolerable). Evaluated on a Xilinx ZYNQ7020 and ZYNQ7045 boards running a quantized AlexNet at different block sizes, the proposed pruning keeps accuracy under 2% below the not-pruned CNN. After pruning, inferences on the ZYNQ7020 and ZYNQ7045 boards achieved 240 and 775 frames per second, respectively.

Sui et al. proposes yet another pruning method tailored to FPGA acceleration (SUI et al., 2023). The KRP pruning also assumes groups of weights within convolutional filters to prune. However, in KRP case, a whole row of weights (e.g., 3 weights are pruned in a 3x3 filter). In their implementation, the PE width is given by the filter width so, by pruning out a row, we can skip all computations without introducing any idleness, creating what the authors call a regular sparse weight matrix. We note that this approach is useful since for some CNNs all layers have filters of same size (e.g., 3x3 filters in VGG and ResNets). KRP also uses the $l1$-norm to select which rows are going to be pruned. After pruning, the authors re-train the CNN with a special learning rate (LR) tracking methodology. In general lines, the LR tracking mimics the LR behavior during the training of the original CNN that was recorded (tracked). The authors report that re-training pruned CNN with LR tracking outperformed state-of-the-art pruning methods: 0.22% accuracy drop at 70% pruning rate for VGG-16 and only 0.01% drop for ResNet-56 at 63.8% pruning rate.

Table 3.2 summarizes the state-of-the-art works presented here on pruning. Note that the table presents both GPU and FPGA works. It becomes clear the wide range of implementations and optimization targets covered by the works on pruning. Also relevant to note is the lack of dynamic pruning methods for FPGA-based inference processing among the evaluated works.

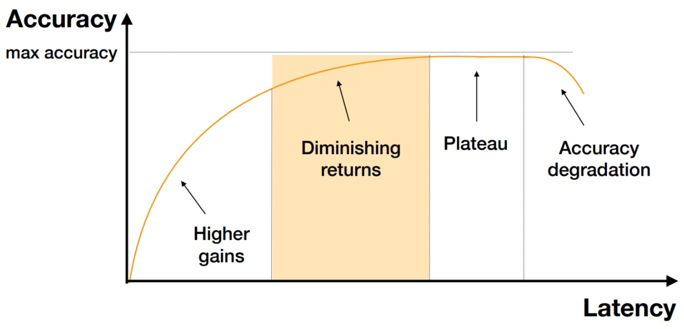
Table 3.2 – Comparison among selected state-of-the-art works on pruning. $\uparrow$ for improvement and $\downarrow$ for reduction on speed-up (SU), model size (MS), and energy (EN).

Work	Platform	Baseline	Main Result	Type of Pruning
Fang, Zeng and Zhang (2018)	Samsung Galaxy S8	Static Pruning	$\uparrow 2\times$ TH	Filter, Dynamic
Kang, Kim and Park (2019)	NVIDIA Jetson TX2	Original CNN	$\downarrow 14\%$ EN	Filter, Dynamic
Xu et al. (2019)	Commercial Smartphones[3]	NetAdapt(YANG et al., 2018)	$\downarrow 21\%$ EN	Filter, Static/Dynamic
Lou et al. (2021)	Jetson Xavier NX	AutoSlim-MnasNet (YU; HUANG, 2019)	$\uparrow 3.5\times$ SU	Sub-netwrks, Dynamic
Xu et al. (2020)	Google Nexus 5	Original CNN	$\uparrow 44\%$ SU	Group, Dynamic
Faraone et al. (2018)	Xilinx KU115	Original CNN	$\uparrow 2\times$ SU	Filter, Static
You and Wu (2020)	Xilinx Zynq ZC706	Deep Compression (HAN; MAO; DALLY, 2016)	$\uparrow 1.3\times$ SU	Unstructured, Static
Véstias (2021)	Xilinx XC7Z020	Original CNN (GUO et al., 2018)	$\uparrow 4.1\times$ SU	Block, Static
Sui et al. (2023)	Xilinx ZYNQ XC7Z035	Not pruned, Full Precision	$\downarrow 27\times$ MS	Mixed, Static

3.1.3 Early-Exit

(FANG et al., 2020) and (LASKARIDIS; VENIERIS et al., 2020) present frameworks for optimizing and deploying early-exit models on embedded GPUs. FlexDNN (FANG et al., 2020) automatically converts regular CNNs to early-exit models with depthwise separable convolution (CHOLLET, 2017) as a building block for the early exits. To place and configure the exits, FlexDNN runs an architecture search that starts with a 'big' exit configuration (i.e., three depthwise separable convolutional layers, each followed by one activation layer, two pooling layers, and one fully-connected layer). Then, FlexDNN iteratively removes a convolutional layer if it causes no drop in the exit rate of that particular branch. Exits are jointly trained in FlexDNN and the entropy as the confidence metric ($Conf(y) = 1 + 1/logC \sum_{c \in C} y_c log y_c$). Overall, authors report up to 7% *increase* in accuracy and as much as 4.2$\times$ energy reduction against the original CNNs.

Figure 3.2 – Accuracy-Latency trade-off on early-exit CNNs (LASKARIDIS; VENIERIS et al., 2020).



HAPI proposed by Laskaridis, Venieris et al. (2020) is a framework for adding early exits to CNNs. First, HAPI builds the input CNN as a Synchronous Dataflow (SDF) graph, enabling fast modeling of the backbone and exit layers. With the SDF representation, the design space can be cast as a Multi-Objective Optimization problem (taking into account the model accuracy, latency, and memory footprint) restricted by user-defined latency and storage upper bounds. The modeling carried out in HAPI helps designers to navigate the accuracy-latency (see Figure 3.2) in a formally defined trade-off. HAPI aims at delivering designs in the highlighted area of Figure 3.2, where latency is not unnecessarily increased (with no accuracy gain). HAPI is evaluated on VGG-16, ResNet, and Inception-V3 CNNs against SoTA early-exits (Branchynet (TEERAPITTAYANON; MCDANEL; KUNG, 2016) and SDN (KAYA; HONG; DUMITRAS, 2019b)) and against other hand-optimized networks, the MSDNet (HUANG et al., 2017a) and MobileNetV2 (SANDLER et al., 2018). HAPI is shown to deliver inferences with accuracy higher than SDN and Branchynet on all evaluated Service-level Agreements (SLA) with accuracy up to 55% higher than Branchynet. Against hand-optimized CNNs, HAPI also shows improved accuracy with speedups of up to $5.11\times$ over the MobileNetV2 CNN.

3.1.3.1 Early-Exit CNNs on FPGA

On FPGA, Farhardi et. al propose a reconfiguration-based method of execution flow for early-exits, called Adaptive and Hierarchical CNN (AH-CNN) (FARHADI; GHASEMI; YANG, 2019). Using a fairly small FPGA device (Xilinx Zynq-7000), AH-CNN can run a large ResNet-18 CNN configured with three exits (two early). The AH-CNN is built on top of a feedback procedure that for each not-taken exit, it reconfigures the FPGA with the following layers (from the next backbone layer after the branch). In the AH-CNN inference procedure, Farhadi et al. also propose a modified confidence threshold. The confidence (taken as the exit TOP-1 softmax) is compared to a different threshold depending on the highest-ranking class. Classes (labels) can be given a certain priority that translates as an increased threshold. Evaluated on CIFAR-10, -100, and SVHN datasets, AH-CNN can reduce by up to 57% the average ResNet computational cost.

DynExit (WANG et al., 2019) approaches early-exit on FPGA from two fronts: first, it presents a tool for tuning the training of early-exit ResNets and, second, it offers a single-engine FPGA accelerator for accelerating these multi-branch CNNs. For the first part, Wang et al. optimized the loss-weights in the Joint Loss Function (recall that the weighted loss from all all exits are accumulated as a single join loss - see w_n in Equation 2.7 in Section 2.1.3.1), improving the early-exit final accuracy (1.20% on average across four ResNet models on two datasets). This optimization was made by replacing the fixed weights w_n per trainable weights. In this way, each w_n gets trained *together* with the CNN weights. On the hardware side, Wang et al. applied a series of transformations to reduce the cross-entropy function (that evaluates the exit confidence) to a pair of one exponential and one natural logarithmic operations. These two operations get synthesized to hardware and added to the single-engine accelerator so the exit decision gets implemented on the FPGA. Running at 220MHz on a Zynq-7000, the ResNet110 loses only -0.07% on accuracy.

Kong et al. used FINN to compile early-exit CNNs to FPGA (KONG; NUNEZ-YANEZ, 2022; KONG; NIKOV; NUNEZ-YANEZ, 2022). The authors added a branch to a quantized AlexNet CNN. Kong et al. used a no longer supported FINN architecture called Multi-Layer Offload, that implements a single-engine accelerator. This single-engine accelerator processes convolutional layers one-by-one, and, for the fully-connected layers, those are either processed in the co-processor in one of the authors implementation (KONG; NIKOV; NUNEZ-YANEZ, 2022) or on the FPGA accelerator (KONG; NUNEZ-YANEZ, 2022). After being trained and quantized with Brevitas, the CNN was exported to ONNX and compiled. Authors used entropy ($\sum_{l=1}^{L} y_l log_2(y_l)$) to measure the branch confidence. When compared to the original AlexNet on FPGA, authors report a 1.56% accuracy drop at speed-ups of around 20% in (KONG; NIKOV; NUNEZ-YANEZ, 2022) and 1.52$\times$ in (KONG; NUNEZ-YANEZ, 2022). Authors also offered a comparison against CPU (i5-9300H) and GPU (RTX2060), where the FPGA-based (a PYNQ-Z2 board) implementation achieved 1.22$\times$ and 0.79$\times$ speed up, respectively. Regarding the slow-down over the GPU, it is important to note that the GPU used has TDP of 160W, dozens of times higher than the low-power PYNQ-Z2 board.

Atheena is a toolflow for automating the synthesis of early-exit CNNs on FPGA (BIGGS; BOUGANIS; CONSTANTINIDES, 2023). It extends the fpgaConvNet (VENIERIS; BOUGANIS, 2016) framework that compiles CNNs to dataflow accelerators. Atheena focuses on the hardware implementation and, thus, it takes as input a CNN with exits already placed and configured. Atheena has two main tasks: profiling and optimization. First, the framework will run the early-exit CNN on a profiling dataset to profile the exits on the exit probabilities and accuracy. The goal of this profiling is to find a probability p for which an input will exit at each of the CNN exits. These probabilities will help Atheena optimizer to allocate the FPGA resources to the exits. Atheena extends the Throughput-Area Pareto (TAP) function from fpgaConvNet that, constrained by the FPGA resources (i.e., BRAM, LUT, DSP, and FF), optimally allocates resources to each layer in the dataflow accelerator so that throughput is optimized. Now that not every layer is used by every input (e.g., backbone layers for an early exited input), Athenna combines two TAPs (one for each branch). This combination is implemented by *scaling* (by their exit probability p) the resources of each branch. Atheena achieves speedups of up to 2.78× over no early-exit CNN running on fpgaConvNet accelerators.

Table 3.3 summarizes the state-of-the-art works on early-exit (targeting both GPU and FPGA platforms). It becomes clear the accuracy advantages of early-exit. By adding early-exit to the traditional CNNs, it is possible to accelerate inference processing (through the early-exit intrinsic runtime adaptation) while the accuracy is either improved or suffer small drops only.

Table 3.3 – Comparison among selected state-of-the-art works on early-exit. $\uparrow$ for improvement and $\downarrow$ for reduction on speed-up (SU) and FLOPs (FL).

Work	Platform	Baseline	Main Result	Accuracy Impact
Fang et al. (2020)	NVIDIA Jetson Xavier	Original CNN	$\uparrow$ 4.3× SU	No Impact
Laskaridis, Venieris et al. (2020)	Nvidia Jetson Xavier	Original CNN	$\uparrow$ 5.11× SU	$\uparrow$ 2.45%
Farhadi, Ghasemi and Yang (2019)	Xilinx Zynq-7000	SkipNet (WANG et al., 2018)	$\downarrow$ 57% FL	No Impact
Wang et al. (2019)	Xilinx Zynq-7000	Original CNN	$\downarrow$ 43.6% FL	$\downarrow$ 0.07%
Kong and Nunez-Yanez (2022)	Xilinx PYNQ-Z2	Original CNN	$\uparrow$ 1.52× SU	$\downarrow$ 1.64%
Biggs, Bouganis and Constantinides (2023)	Xilinx ZC706	Original CNN	$\uparrow$ 2.78× SU	No Impact

3.1.4 Offloaded Inferences

Systems like DjiNN (HAUSWALD et al., 2015) and Clipper (CRANKSHAW et al., 2017) pioneered inference serving (or offloading). DjiNN makes a set of trained DNN models available in multi-GPU servers to process inferences requests arriving from end-nodes. These requests come through a TCP/IP socket protocol developed by the authors. When a request arrives, "DjiNN spawns a worker thread, executes the DNN computation, and sends the prediction back to the application" (HAUSWALD et al., 2015). Their serving approach achieves a near-linear scaling for three of the seven applications evaluated (from image to language processing), with throughput improvements of over 100 × over integrated GPU servers.

Clipper (CRANKSHAW et al., 2017), on the other hand, not only serves inferences from a "bag-of-models" but also employs a model selection mechanism to fuse the output of parallel CNNs based on the application feedback (i.e., more than one DNN for serving each task). Besides that, Clipper introduced model caching, so it does not need to re-evaluate the model (latency, throughput, etc.) on every inference request. The information collected and cached is used to inform the model selection at runtime. Clipper also proposes adaptive batching. Users can specify a Service Level Objective (SLO) with a minimum tolerable latency that will be used to tune the batched inference requests (increasing the server response latency allows for collecting more requests on the same task and extracting more parallelism from the GPUs running the inferences).

When it is possible to run DNNs locally as well as offload them, there is a need to decide on which side (device or server) to process (see Binary Offloading in Section 2.1.3.2). DeepDecision implements a framework (RAN et al., 2018b) that considers the interaction between accuracy, device battery, and the network condition to make the offloading decision a combinatorial optimization problem. DeepDecision is evaluated on an augmented reality application that streams a video constantly for inference. It can act on four optimization knobs (besides the offloading decision). DeepDecision can adjust the frame resolution, switch between two DNN models, and change the video bitrate and the sample rate. DeepDecision improves the overall responsiveness and average accuracy compared to offload-only or local-only inferences.

On offloading techniques that partition a DNN, we have SPINN (LASKARIDIS et al., 2020) as a representative SoTA work. SPINN leverages early-exit (Section 2.1.3.1) as possible locations for partitioning the inference processing. Therefore, the backbone layers can branch into a local early exit or can be followed by the offloading of that feature map. SPINN offers a two-step workflow with offline and online steps. Before deployment, SPINN adds the early exits to the CNNs, identifies all possible split points, and runs an initial profiler to gather initial latency estimates. At runtime, SPINN uses a scheduler to adjust early-exit and offloading policies based on target SLA and runtime conditions. SPINN also leverages the presence of zero values in the feature map (e.g., after ReLU) to reduce the offloading overhead. Compared to other SoTA works, SPINN improves throughput by up to $2\times$ with an accuracy 20% higher under latency constraints.

Pacheco et al. also optimized the offloading based on the early-exit (PACHECO; COUTO; SIMEONE, 2023; PACHECO; COUTO; SIMEONE, 2021). When we consider both together, the early-exit confidence threshold also acts as a parameter controlling the rate of offloaded inference maps. Now, besides continuing to the backbone layers, these not-confident feature maps get offloaded, incurring additional costs from the communication channel. In their work, Pacheco et al. proposed a "calibration study on different datasets and early-exit DNNs for the image classification task." Particularly, they employ a temperature-scaling calibration method (GUO et al., 2017). This method can be used to tune any DNN parameter and, for early-exit confidence threshold, it is used to tune the "over confidence" problem (i.e., exits outputting a high confidence for wrongly classified inputs). With the temperature scaling the exit i logit vector (output before softmax), $\mathbf{z}_i = (\mathbf{x}|\Theta)$, gets scaled by T, and the final output vector y becomes $y = softmax(\frac{\mathbf{z}_i=(\mathbf{x}|\Theta)}{T})$. Now, T can be calibrated to optimize the early-exit model. Pacheco et al. evaluate a global and per-exit calibration of T that is modeled as a optimization problem minimizing the validation loss. Overall, the authors report that calibrating the early-exit results in classifications of higher accuracy and higher probabilities of meeting latency requirements.

Seifeddine et al. also combined early-exit and offloading optimizations (SEIFEDDINE; ADJIH; ACHIR, 2021). In this work, however, they proposed a ML-based decision - instead of a manual configuration (or calibration) of the confidence thresholds. At each exit, running locally (e.g., at an IoT device), there are three possible outcomes: exit (finish the classification), continue (keep the inference on the backbone layers running locally), or offload (send the intermediate feature map to an edge server). The authors model this decision as a Markov Decision Process (MDP), which observes the current exit index, the exit output vector, and the previous states. Optionally, any other dynamically changing parameter can be included in the decision process such as communication costs. To solve the MDP, the authors use a reinforcement learning algorithm, the Deep-Q Network (DQN). Generally speaking, they propose training a DNN (by reinforcement) to decide the inferences early exits and offloading. The DQN is trained together with the early-exit DNN with each output vector from the early-exit (labeled given its correct/incorrect classification) being served as input to DQN. The authors model the energy cost as processing one block of layers costing exactly 1 energy unit as a proof of concept.

Table 3.4 presents the selected state-of-the-art works on offloading. It can be seen that the state-of-the-art on distribute inference processing shifted from static solutions (of full offloading, always processing at the cloud, first two rows for example) to systems were the decision to split is dynamic and, therefore, both end devices (e.g., IoT) and cloud/edge server are capable of processing inferences.

Table 3.4 – Comparison among selected state-of-the-art works on offloading. $\uparrow$ for improvement throughput (TH) and accuracy (AC).

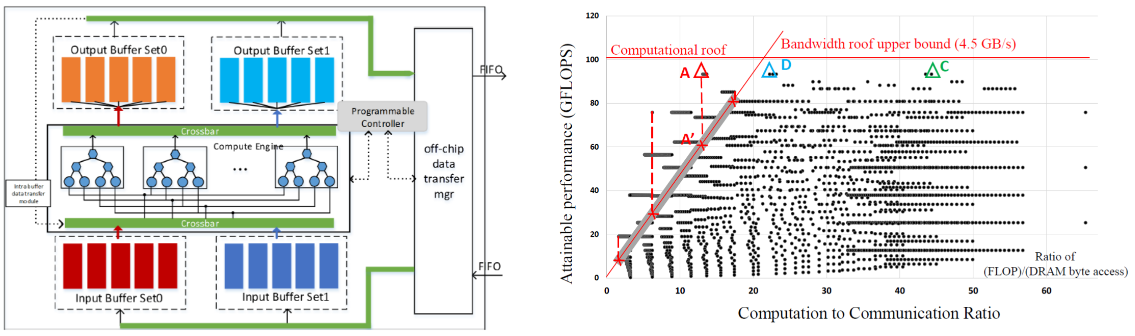
Work	End Device	Server	Type of Offloading	Baseline	Main Result
Hauswald et al. (2015)	-	(integrated set) Nvidia K40	Full Offload	Desegregated GPUs	$\uparrow$ 120$\times$ TH
Crankshaw et al. (2017)	-	Nvidia Tesla K20c	Full Offload	TensorFlow Serving	$\uparrow$ 5.2% AC
Ran et al. (2018b)	Samsung Galaxy S7	Nvidia GTX970	Binary Decision	Local Only (Smaller CNN)	$\uparrow\sim$ 30% AC
Pacheco, Couto and Simeone (2023)	Nvidia Jetson Nano	Amazon EC2	Split, Early-Exit	Not Calibrated Early-Exit	$\uparrow$ AC

3.2 Optimizations at the Hardware Level

3.2.1 High-Level Synthesis

HLS optimizations can be carried out *inside* the HLS tools or *with* the HLS tool (KHANH et al., 2015). Works optimizing inside the HLS tool fall in the CAD development and are unrelated to this thesis. Here, we are interested in works supported by the HLS tool (e.g., tuning HLS directives). Especially, we cover the ones targeting acceleration of DNN workloads.

Figure 3.3 – (a) the accelerator and (b) its roofline plot when executing AlexNet layer 5 (32-bit floating point) (ZHANG et al., 2015a).


Output Buffer Set0
Output Buffer Set1
Programmable Controller
Crossbar
Compute Engine
Intra buffer data transfer module
off-chip data transfer mgr
FIFO
FIFO
Crossbar
Input Buffer Set0
Input Buffer Set1
Computational roof
Bandwidth roof upper bound (4.5 GB/s)
A
A'
D
C
Attainable performance (GFLOPS)
Ratio of (FLOP)/(DRAM byte access)
Computation to Communication Ratio
0
20
40
60
80
100
120
10
30
50

(a) Block diagram of the single-engine accelerator

(b) Design Space in roofline coordinates

An HLS optimization framework for CNN accelerators was proposed by Zhang et al. (ZHANG et al., 2015a). One critical issue in the design of FPGA-based CNN accelerators is the match between computation throughput and the FPGA's memory bandwidth, which may cause either FPGA or bandwidth underutilization. To approach this problem, Zhang et al. propose an analytical roofline model that guides the HLS. Figure 3.3(a) present the single-engine accelerator used and, on its side, the design space (in roofline coordinates) created by exploiting the HLS. These multiple design points are created by tackling compute and memory-related HLS operations. To optimize computations, the authors explore loop unrolling and pipelining across the accelerator HLS kernel. To optimize memory access, authors: apply local memory promotion (POUCHET et al., 2013) to eliminate useless memory access; and, transform loops for data reuse with the help of a polyhedral-based optimization framework (POUCHET et al., 2013). Accelerators synthesized by the authors outperform other SoTA accelerators, achieving over 60 GFLOPS under 100MHz working frequency.

Shen et al. use HLS to implement an accelerator and design flow based on what they call Convolutional Layer Processor (CLP) (SHEN; FERDMAN; MILDER, 2017). Their accelerator works by deploying multiple of these CLP that are "single-engine" designs. Shen, Ferdman and Milder (2017) observe that the performance of a single-engine is not only due its size (e.g., number of PEs in a systolic array), but it also depends on the accelerator compatibility with the convolution parameters (e.g., number and size of filters). Therefore, the authors propose tuning the HLS to optimize each CLP to a (subset of) layer(s) instead of a single traditional one-fits-all single-engine design. To produce a Multi-CLP accelerator, the authors developed an optimization algorithm that, given CNN layer dimensions and FPGA resource budget (as well as the modeling for performance, bandwidth usage, and DSP usage), partitions the FPGA into CLPs. The optimization step defines the HLS pragmas to best match each CLP to one or more layers of the CNN. The authors report that the DSP utilization exceeds 90% (when it can stay below 24% in other SoTa single-engine accelerators). These higher utilization speeds up inference of AlexNet in 3.8× over the same SoTA accelerator on a Xilinx Virtex-7 FPGA.

Wang et al. exploited HLS to developed optimized CNN accelerators on Virtex FPGAs (SUN et al., 2017). Similar to the previous work by Shen et al., the approach taken in this work is also based on the fact that having a single design to perform all layers may lead to significant inefficiencies. This time, instead of deploying parallel accelerators on the same FPGA that are independently scheduled, Wang et al. connect the convolution layer accelerator unit (CLAU) in a pipelined fashion. Each CLAU is in charge of performing a particular layer in the CNN. Before synthesis, each CLAU is optimized in such a way to balance out the CLAUs latency (minimizing the worst-case CLAU which improves the pipeline throughput). This is done through an exhaustive search on the HLS pragmas controlling the CLAU parallelization (i.e., via the HLS pragmas for loop tiling, unrolling and interchange). Additionally, to alleviate the computation and band-with cost of fully-connected layers, Wang et al. prune neurons from those layers. A non-structured pruning is used and, because of that, the FC weights need to be stored in the compressed sparse column (CSC) format. Around 10% of the neurons in the last three FC layers of the AlexNet, the use case, were pruned. The proposed accelerator achieves a peak performance of 498.6 Giga Operations per second (GOP/s) in a Virtex-7 at 100MHz.

Table 3.5 shows the selected state-of-the-art works on using HLS to optimize CNN accelerators. Note that second and third rows use the work in the first row as baseline. The reader may also note, with respect to single-engine, the significant improvement achieved by a dataflow-like accelerator (third row), where the mapping between CNN layer to HLS modules offer a better opportunity for optimization.

Table 3.5 – Comparison among selected state-of-the-art works using HLS to optimize the inference processing. $\uparrow$ for improvement on throughput (TH) and OP/s (OP).

Work	Platform	Type of Accelerator	Baseline	Main Result
Zhang et al. (2015a)	Xilinx VC707	Single-Engine	CNP (FARABET et al., 2009)	$\uparrow$ 11.7$\times$ OP
Shen, Ferdman and Milder (2017)	Xilinx Virtex-7	Parallel Single-Engines	(ZHANG et al., 2015a)	$\uparrow$ 3.8$\times$ TH
Sun et al. (2017)	Xilinx VC709	Dataflow	(ZHANG et al., 2015a)	$\uparrow$ 21$\times$ TH

3.3 Contributions to the State-of-The-Art

Table 3.6 – Comparison against the state-of-the-art works.

	Optimizations							
Works	Quantization	Pruning	Early-Exit	Offloading	HLS	Statically Applicable	Dynamically Applicable	FPGA Applicable
Umuroglu and Jahre (2017)	✓					✓		✓
Amiri et al. (2018)	✓					✓		✓
Neda et al. (2022)	✓					✓		✓
Capotondi et al. (2020)	✓					✓		✓
Coelho Jr et al. (2021)	✓	✓			✓	✓		✓
Xu et al. (2019)		✓					✓	
Lou et al. (2021)		✓					✓	
Fang, Zeng and Zhang (2018)		✓					✓	
Xu et al. (2020)		✓					✓	
Hawks et al. (2021)	✓	✓				✓		✓
Faraone et al. (2018)	✓	✓				✓		✓
You and Wu (2020)	✓	✓				✓		✓
Véstias (2021)	✓	✓				✓		✓
Sui et al. (2023)	✓	✓				✓		✓
Fang et al. (2020)			✓			✓		
Laskaridis, Venieris et al. (2020)			✓				✓	
Wang et al. (2019)	✓		✓				✓	✓
Kong and Nunez-Yanez (2022)	✓		✓			✓		✓
Kong, Nikov and Nunez-Yanez (2022)	✓		✓			✓		✓
Biggs, Bouganis and Constantinides (2023)	✓		✓		✓	✓		✓
Hauswald et al. (2015)				✓		✓		
Crankshaw et al. (2017)				✓		✓		
Ran et al. (2018b)				✓			✓	
Laskaridis et al. (2020)			✓	✓			✓	
Pacheco, Couto and Simeone (2023)			✓	✓		✓		
Seifeddine, Adjih and Achir (2021)			✓	✓		✓		
Zhang et al. (2015a)					✓	✓		✓
Shen, Ferdman and Milder (2017)	✓				✓	✓		✓
Sun et al. (2017)	✓				✓	✓		✓
This Thesis	✓	✓	✓	✓	✓	✓	✓	✓

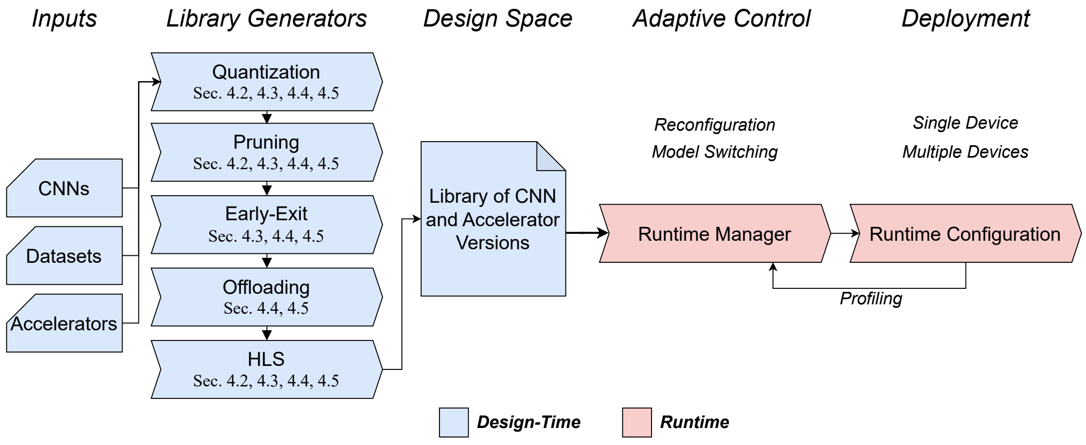
There is no winner today for optimizing the inference processing (i.e., how to reduce the processing/energy/resource requirements at little or no quality impact). On top of that, the IoT-Edge continuum presents us with a highly dynamic environment that demands adaptability due to its varying workload, unpredictable network, etc. This lack of a united approach can be noticed in Table 3.6 that brings the state-of-the-art (SoTA) works discussed so far. Table 3.6 identifies the optimizations covered by the authors and whether their solutions are static, dynamic, or FPGA-applicable. Therefore, this thesis *general contribution* lies in, first, covering the inference optimization problem in a multi-level approach (from FPGA to CNN) creating a **unified design space**. Second, we extend the first contribution by exploiting this design space dynamically, providing **runtime adaptation** of the inference processing.

This thesis also advances the SoTA with other complementary contributions. For instance, we have that, unlike works exploring pruning on GPU or CPU platforms, we can take the hardware platform into account as an additional optimization axis when pruning for FPGA (e.g., with parameterizable accelerators). In this regard, we contribute to the pruning SoTA with a new method that, besides the CNN, considers the accelerator properties to prune weights. Additionally, we will also advance the SoTA on FPGA-based dataflow accelerators by taking these pruned CNNs and enabling their switching at runtime (with no need for FPGA reconfiguration), which was not possible before (due to all SoTA CNN-to-FPGA tools generating accelerators hard-wired to CNNs - e.g., no dynamic pruning on FPGA in Table 3.6). Another SoTA advancement from this thesis regards the early-exit, in which we were the first to combine early-exit with the statically applied optimizations of pruning and quantization (novel among all hardware platforms). For example, during the development of this thesis, we foremost reported that prematurely exiting can even improve the accuracy of heavily pruned CNNs. Finally, this multi-level optimization also presents a novelty in the offloading SoTA. In this context, this thesis is the first to propose an offloading where CNNs get partitioned, and the partitions can be optimized (i.e., pruned, quantized, etc.) according to the node's requirements. The next chapter presents this thesis's framework for delivering such contributions.

4 GENERAL WORKFLOW

As explained at the beginning of this document, our goal is to build a framework that (i) develops, evaluates, and combines optimizations at hardware and CNN levels across the IoT-edge continuum; and, (ii) exploits those optimizations for runtime adaptation of the inference processing according to the current environment and application requirements. To that end, we have explored different types of FPGA accelerators, CNN models, datasets, and optimizations. We point out that not all works carried out in this thesis are presented in this chapter. Particularly, at the beginning of this work, we used the CHaiDNN-v2 (Xilinx Inc, 2020) accelerator, a single-engine accelerator that was open-sourced by Xilinx. Since we moved to a more modern accelerator later, we opted to leave the works using CHaiDNN-v2 in the appendix chapters. Those initial works also included the pruning (Appendices A and B) and approximate computing (Appendix D) optimizations and evaluations with images with higher resolution (Appendix C). In the remaining of this chapter, we present the works that directly supported this thesis' final contribution (which is presented in the last section of this chapter).

Figure 4.1 – The general workflow taken throughout this thesis.



Throughout the development of this thesis, we have structured our work as the workflow in Figure 4.1. The workflow follows from left to right, crossing the design-time steps of Inputs, Library Generators, and Design Space (blue shapes) to the steps taken at runtime, namely Adaptive Control, and Deployment (red shapes). The ***Inputs*** step receives the original CNNs, datasets, and accelerators' code. Those will feed the optimizations that work as ***Library Generators*** to create the multiple versions of CNNs and accelerators. These versions are grouped in the Library, constituting our ***Design Space*** step. The library works in between the design-time and runtime phases, and supports the runtime adaptability. The ***Adaptive Control*** is performed by our Runtime Manager that searches the library at runtime and, based on a system profiling, selects the Runtime Configuration that best adapts to the current application environment. The Runtime Configuration is used at the ***Deployment*** step and dictates how and where the inference processing will be executed (i.e., the deployed level of each optimization, like the quantization bit-width, pruning rate, offloading location, etc.).

Over the development of this thesis, we have split the work into packages, which we called *workfronts*, each focusing on one (or a subset) of the optimizations in Figure 4.1. There are four workfronts in this chapter (Subsections 4.2, 4.3, 4.4, and 4.5 also indicated in Figure 4.1). These workfronts were build on top of each other and started targeting the pruning of CNNs. Then, early-exit was added and, finally two types of offloading were considered. We note that all workfronts use quantization and HLS, which are part of the original FINN/Brevitas design-flow, and, thus, are not presented as the main contributions of the workfronts. By doing that, the optimizations could be integrated into a single multi-level optimization framework - this is the last workfront presented in Section 4.5.

In the following subsections, we first present the tools and an overview on the methodology used across the work. Later, the workfronts are presented in greater detail with their respective results. Each workfront will specialize the base workflow in Figure 4.1 with the used inputs, optimizations, and particular configurations.

4.1 Methodology Overview

In this section we present the FPGA accelerator (with the evaluated boards and tools), datasets, and CNN that were used. We leave to each workfront to detail their particular implementation (e.g., configurations and implementations particular to each optimization) and the employed evaluation scenario.

4.1.1 Accelerators

As pointed out in the Inputs step in Figure 4.1, the user is in charge of feeding the accelerator. As mentioned before, this thesis explored the CHaiDNN-V2 single-engine accelerator first. Only afterwards, we shifted to the **FINN (BLOTT et al., 2018)** accelerator. First, because it has a stronger support from Xilinx (CHaiDNN was discontinued) and has a well-defined design flow including the Brevitas framework for training quantized DNNs. Second, its dataflow architecture offers more optimization opportunities. Accelerators used across our dataflow experiments were synthesized within the Xilinx's FINN design flow with Vivado 2019.2 targeting either a Xilinx Zynq Ultrascale+ MPSoC ZCU102 board (XCZU7EV) or a Xilinx Pynq-Z1 board (XC7Z020), both at 100MHz. We used Xilinx Vivado for resource usage and power extraction and Verilator RTL simulations for performance (all executed within FINN's infrastructure).

4.1.2 Datasets and CNNs

The other two user inputs in Figure 4.1 are the original CNNs and the datasets for training. We have used the following datasets in our evaluations:

- The **CIFAR-10** dataset (KRIZHEVSKY; HINTON, 2009). This dataset contains 50,000 training and 10,000 test images. The classification task consists of 10 classes ranging from "airplane" to "bird," and "truck."
- The German Traffic Sign Recognition Dataset (**GTSRB**) (HOUBEN et al., 2013), which contains 39,209 training and 12,630 test images of 43 classes representing road signs like speed limits and stop signs.

Unless stated otherwise, all images consider CIFAR-10's image resolution (3x32x32). Accuracy results are reported on Brevitas TOP-1 test accuracy. All retraining (after pruning) is performed for 40 epochs (LI et al., 2017). Early-exit CNNs are trained following the hyper-parameters from (TEERAPITTAYANON; MCDANEL; KUNG, 2016), with 1.0 weight on the first exit and 0.3 to the remaining. All training uses standard data augmentation (padding, crop, and flipping) and learning rate of 0.001 with decay of 0.1. Retraining was executed on Intel Xeon E5-2640 with NVIDIA Tesla K20m GPU.

We used the **CNV** CNN for evaluation:

- CNV is a VGG-like CNN available in the Brevitas (PAPPALARDO, 2023) repository. CNV has three blocks, each with a sequence of two CONV layers (3x3 filter) of 64, 128, and 256 channels, respectively. First and second blocks are followed by 2x2 max-pooling layer. The last block is followed by two 512-neuron FC layers plus a last one with the number of neurons equal to the number of classes. CNV can be quantized to any bit-width with Brevitas. Here, we quantize CNV at 1, 2, and 4 bits[1].

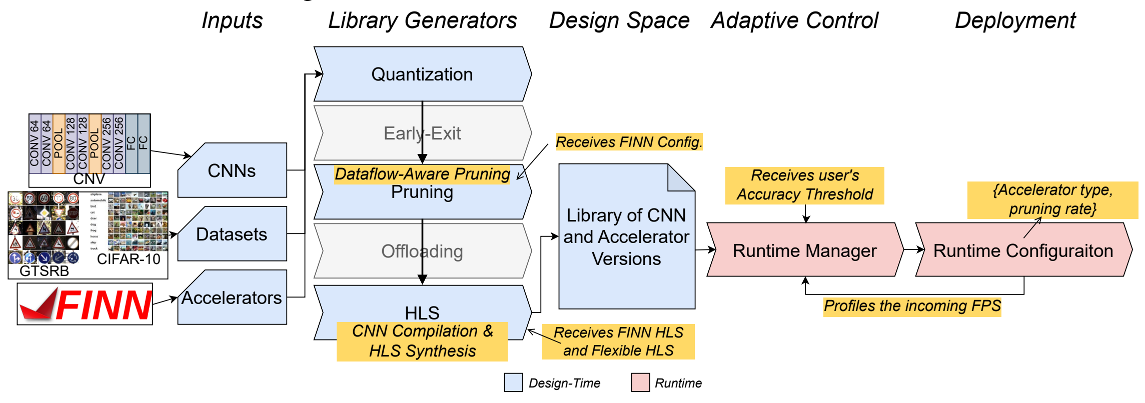
4.2 Workfront 1: Pruning

One issue of the FPGA-based dataflow accelerators (e.g., FINN) is that they are specifically synthesized for one CNN model, precluding fast model switching (e.g., changing pruning rate at runtime is not possible). In other words, if one requires model switching on top of FINN, frequent use of FPGA reconfigurations to switch from one accelerator to another will be needed. Meaning that the inference processing needs to stop in order to switch between CNNs.

Therefore, in this workfront, we targeted enabling shorter switching times in dataflow accelerators. Specifically, we implemented an extra level of adaptability on top of a SoTA dataflow (the FINN framework), so pruning and fast model switching are possible. As we are going to see next, however, providing runtime adaptability in dataflow accelerators requires extra logic and power, creating a trade-off between flexibility and efficiency for these FPGA accelerators. Because of that, this workfront will make use of *flexible-pruning* accelerators (i.e., allow for fast switching), proposed here, and accelerators fixed to particular pruned models (*fixed-pruning*), synthesized with the original FINN design-flow.

4.2.1 AdaFlow

Figure 4.2 – Workflow for the AdaFlow workfront.



[1]in the Brevitas/FINN nomenclature, CNVWxAy specifies a CNV with weights quantized to x bits and activations to y bits.

We called this workfront AdaFlow for adaptive inference on dataflow accelerators (KOROL et al., 2022). Figure 4.2 presents its workflow. It specializes the base workflow from Figure 4.1 in the following aspects:

- At the design-time Inputs steps, AdaFlow uses the FINN accelerator with CNV CNNs on the GTSRB and CIFAR-10 datasets;
- Under the Library Generators step, AdaFlow uses Pruning (as well as Quantization and HLS). AdaFlow is evaluated with CNV quantized at two bits for activation and one or two for weights (i.e., CNVW2A2 and CNVW1A2). Notably, it includes a new pruning mechanism (Dataflow-Aware Pruning) and an HLS synthesis flow that extends the FINN framework (CNN Compilation & HLS Synthesis). These two optimizations work on the quantized CNN, and receive from the user the FINN configuration files (FINN Config. in Fig. 4.2) and the set of our newly implemented HLS classes (Flexible HLS in Fig. 4.2);
- At runtime, AdaFlow Runtime Manager receives an Accuracy Threshold from the user to select Runtime Configurations (which now defines the current accelerator type, fixed or flexible, and the current pruning rate).

More specifically, in the AdaFlow workfront, the library not only stores the pruned CNN models, but also stores the flexible dataflow accelerators with support fast model switching (that use our ‘Flexible HLS’) or the ones that are fixed and require model switching via FPGA reconfiguration (synthesized with the traditional FINN, ‘FINN HLS’ in Figure 4.2). At the Adaptive Control step, the runtime manager automatically chooses the best Runtime Configuration (i.e., pruning rate and accelerator type) from the library to dynamically adapt the inference processing according to the profiling and the user’s Accuracy Threshold. Next, we detail the optimization steps particular to AdaFlow’s Library Generators step.

4.2.1.1 Library Generators

Pruning Optimization. To provide multiple design points on the accuracy-resource trade-off for a single dataflow accelerator, we propose a filter pruning mechanism called Dataflow-Aware Pruning. Besides the CNN model, this pruning mechanism takes into account properties of the dataflow accelerator. In FINN, there are two main constraints refraining a dataflow accelerator from loading a CNN model that had its CONV layers *freely* pruned: for each MVTU performing a CNN layer, we have that the number of

Table 4.1 – Feature Map (FMAP) shapes (channel x height x width) and the resulting size in number of words (#words) before and after pruning at 50% pruning rate.

	Original		Pruned	
	FMAP	#Words	FMAP	#Words
Input	3x32x32	3072	3x32x32	3072
CONV 1	64x30x30	57600	32x30x30	28800
CONV 2	64x14x14	12544	32x14x14	6272
CONV 3	128x12x12	18432	64x12x12	9216
CONV 4	128x5x5	3200	64x5x5	1600
CONV 5	256x3x3	2304	128x3x3	1152
CONV 6	256x1x1	256	128x1x1	128
FC 1	512x1x1	512	512x1x1	512
FC 2	512x1x1	512	512x1x1	512
FC 3	10x1x1	10	10x1x1	10

CONV filters (or neurons, in the case of a fully-connected layer) has to be divisible by the number of PEs; and, the number of SIMD lanes has to be divisible by the number of input channels (recall Background Section 2.2.2.2). Such constraints guarantee the correctly feeding of all PEs and SIMD lanes, ensuring full parallelism (i.e., no idle PEs or SIMD lanes). To build models that respect such constraints, we implemented the Dataflow-Aware Pruning mechanism, which, starting from an initial CNN model, prunes filters to generate a pruned model version with particular accuracy and resource profile.

For each pruned model, our Dataflow-Aware Pruning takes an initial CNN model, a FINN configuration file (containing the dataflow parameters), and a pruning rate (percentage specifying how many filters to prune). Then, for every CONV layer, the procedure attempts to prune a certain amount of filters r_i in such a way that it respects the

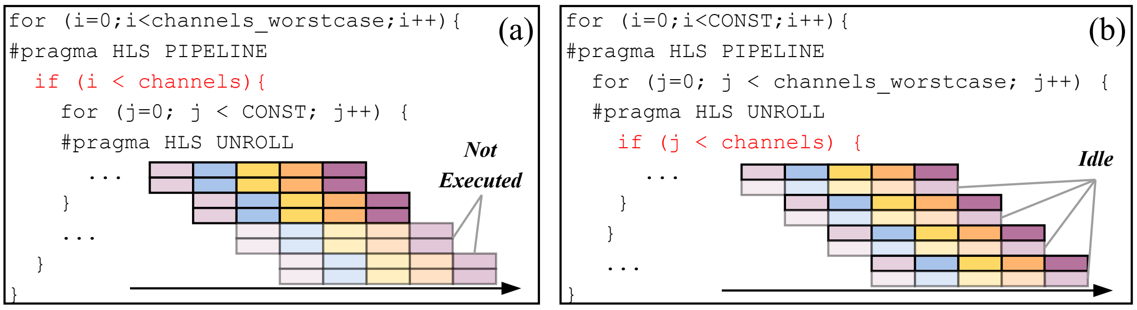
$$(ch_{out}{}^{i} - r_i)\, mod\, (PE_i) = 0 \text{ and } (ch_{out}{}^{i} - r_i)\, mod\, (SIMD_{i+1}) = 0$$

constraints, where PE_i and $SIMD_{i+1}$ give the MVTU's number of PEs and SIMD lanes (see Figure 2.15(b)) of current i and next layer $i+1$, respectively. $ch_{out}{}^{i}$ gives the not-pruned number of channels for that layer (from the initial CNN). If the constraints are not met, the procedure iteratively decreases r_i until they are met. Dataflow-Aware Pruning leverages the filter selection proposed in (LI et al., 2017) that measures the relative importance of a filter in each channel by calculating, from the floating-point representation, the sum of its absolute weight values (ℓ_1-norm). After all CONV layers have been pruned, the model can be retrained and exported as an ONNX file. We call the dataflow accelerators synthesized from pruned CNN models Fixed-Pruning since they can only execute that particular model. AdaFlow's pruning is implemented on top of Brevitas (PAPPALARDO, 2023). Here, the pruning rate is ranged at fixed steps for each dataset/initial CNN model to generate the multiple versions. For example, Table 4.1 shows the feature map sizes in CNV (the CNN used across our experiments) before and after a 50% pruning rate. The reader may note that with the Dataflow-Aware Pruning only the convolutional layers are pruned, while input (RGB image) and FC layers are left with original shapes (maxpool layers not shown in the are also not pruned since those do not contain any weights).

HLS Optimization. The ONNX files (pruned CNNs) created in the Dataflow-Aware Pruning, are fed to the *CNN Compilation & HLS Synthesis* to synthesize their respective accelerators: Fixed-Pruning (one for each pruned CNN model, using fixed modules - original FINN HLS classes) and Flexible-Pruning (one for all pruned models of the same initial CNN model). For implementing the Flexible-Pruning, it was necessary to create a new set of HLS modules as new C++ HLS classes in FINN, which we called Flexible HLS classes.

The Flexible HLS classes have the capability of switching the CNN without requiring FPGA reconfigurations. For that, we modified FINN's HLS classes with runtime-controllable parameters. Since the pruning technique used by AdaFlow affects only the number of CONV filters and its respective number of output channels, the number of channels is the only parameter that needs to be configured at runtime in the dataflow accelerator. In that sense, Flexible HLS templates differ from regular FINN HLS templates only in the loops in which bounds are affected by the number of CONV channels. Then, at runtime, it can be configured to process a smaller number of channels. Therefore, flexible accelerators are synthesized to the worst case in terms of model size, given by the initial, not-pruned, CNN model. Below, we focus on the difference between Flexible and FINN HLS templates. The modifications made to FINN's HLS classes can be divided into two cases: when the runtime-controllable parameter (i.e., number of CONV channels) affects HLS *unroll* directive and when it affects the *pipeline* directive.

Figure 4.3 – Loops with variable control (red shaded) in Flexible HLS templates with `channels` half the `channels_worstcase` in both examples.



```
for (i=0;i<channels_worstcase;i++){
#pragma HLS PIPELINE
  if (i < channels){
    for (j=0; j < CONST; j++) {
    #pragma HLS UNROLL
      ...
    }
    ...
  }
}
```

```
for (i=0;i<CONST;i++){
#pragma HLS PIPELINE
  for (j=0; j < channels_worstcase; j++) {
  #pragma HLS UNROLL
    if (j < channels) {
      ...
    }
  }
  ...
}
```

We bring two representative examples for presenting each case: the MVTU module, responsible for executing all convolutional and fully-connected layers, and the MaxPool module. MVTU's unroll is independent of the runtime-controllable parameter (MVTU unroll is given by the number of PEs and SIMD lanes - recall Figure 2.15 in Subsection 2.2.2.2). On the other hand, MaxPool unrolling depends on the number of channels. Figure 4.3 presents these two examples, where `channels_worstcase` gives the full number of input or output channels (given by the not pruned model) and `channels` is the runtime-controllable parameter that gives the current number of channels (particular to the currently loaded CNN model version, variable between models). Red shaded `if` statements give the runtime-controllable behavior to both modules. In Figure 4.3(a) we have a simplified version of the MVTU, which is unrolled on a fixed parameter `CONST` (i.e., PE/SIMD values). In this case, the runtime-controllable parameter only affects the pipeline feeding, causing fewer pipeline iterations and a shorter execution time. On the other hand, for the MaxPool module that is unrolled on the runtime-controllable parameter, the loop has to be synthesized to the worst case, and, when `channels` < `channels_worstcase`, some of the units performing the unrolled operation will not be fed, as depicted in Figure 4.3(b).

Information on the number of channels of every model's layer is attached to the model description when AdaFlow prunes a CNN model. Then, during inference, at runtime, the number of channels can be passed to every flexible module in the dataflow (those have an extra 16-bit interface port to set the runtime-controllable, `channels`, parameter). However, the extra circuitry enabling flexible execution creates a small overhead in both resource usage (making the accelerator larger) and performance (increased latency). This trade-off between accelerators' flexibility and efficiency is exploited by AdaFlow to process inferences at the highest levels of adaptability.

We added the set of new flexible HLS template classes to the original FINN framework. Also, FINN's design steps were modified to integrate the new runtime-controllable functionalities and ensure synthesis and correct processing of the Flexible-Pruning accelerators. More implementation details on this workfront are given in Appendices F and G for the hardware and software aspects, respectively.

With CNNs and accelerators generated, a library in the form of a table with a list of pruned CNNs (rows) with their accuracy (extracted after pruning) as well as the throughput values (extracted during synthesis) is created (Design Space step in Figure 4.2). After this step, the runtime phase at the Adaptive Control step takes place.

4.2.1.2 Adaptive Control

Algorithm 2 Runtime Manager search in AdaFlow

```
1: Set $P$ to the priority list of optimization parameters
2: Set the workload $FPS_{in}$, in frames per second, profiled at runtime
3: Set $LastSwitch$ to the time in seconds since the last model switch
4: Set the accuracy threshold $AccTh$ to the value informed by the user
5: Set $Configs$ to the list of all configurations in the Library
6: Set $AccelCriteria$ to the value informed by the user for selecting the accelerator type
7: if $LastSwitch > AccelCriteria$ then
8:     Remove all configurations with Flexible Accelerators from $Configs$
9: else
10:    Remove all configurations with Fixed Accelerators from $Configs$
11: Remove configurations with accuracy below $AccTh$ from $Configs$
12: Remove configurations with throughput below $FPS_{in}$ from $Configs$
13: Initialize $c'$ with any configuration from $Configs$
14: for parameter $p$ in $P$ do // Lexicographic Optimization on the priority list $P$
15:    $min\, p(c), \forall c \in Configs$
16:    s.t. $p_i(c) \leq p_i(c')$, for optimization parameters before $p$ in $P$
17:    Update $c'$ with $c$
18: Return $c'$ as the selected configuration
```

Runtime Manager. The Runtime Manager in the AdaFlow workfront is in charge of selecting the Runtime Configuration from the library produced at design-time (Figure 4.2). Runtime Configurations define the pruned CNN models and accelerator type (Fixed or Flexible). The Runtime Manager will act every time there is a change in either accuracy threshold (set by the user) or incoming FPS (that can be flagged by performance monitors added to the software in charge of the incoming inferences). Algorithm 2 presents the Runtime Manager search. Broadly speaking, it occurs in two main steps. First, it rules out some 'invalid' configurations from the library by selecting the accelerator type (lines 7-10), minimum accuracy (line 11), and minimum FPS (line 12). Only then, the actual search on the remaining configurations takes place (lines 14-17).

Particularly, the Runtime Manager algorithm between lines 7-10 uses a rule-based criteria for selecting accelerator type ($AccelCriteria$ in Algorithm 4.2): Fixed-Pruning accelerators are only selected when models need to be switched at intervals greater than this predefined value, which can be fine-tuned depending on the application and FPGA at hand. For example, if the user sets $AccelCriteria$ to 1 second, the Runtime Manager will only select Fixed-Pruning if, at least, 1 second has passed since the last model switch. Otherwise, Flexible-Pruning accelerators are used. It is important to note that fine-tuning this criteria is possible for other applications.

After the accelerator type is defined, and configurations with accuracy and throughput below the minimum accepted are removed (below Accuracy Threshold and incoming FPS, respectively), the search on the Library starts. To that end, a straightforward *Lexicographic Optimization* search takes place (lines 13-17). For AdaFlow, this search assumes the throughput as the first priority, accuracy as the second (i.e., a $P = \{Throughput, Accuracy\}$ is set as the optimization priority in line 1). Then, the last selected configuration (c') is returned to configure the inference processing.

4.2.2 Results

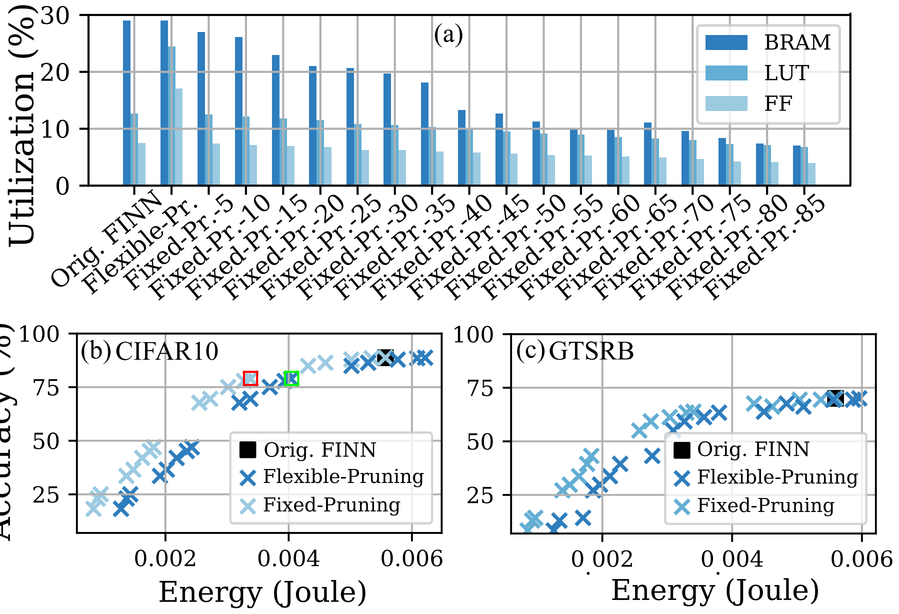
4.2.2.1 Evaluation Scenario

In this workfront, we use the FINN accelerators (as detailed earlier in Section 4.1) with CNVW2A2 and CNVW1A2 CNNs. AdaFlow generates 18 models for each initial CNN with pruning rates from 0% (not-pruned) to 85% (5% steps). Each model generates a *Fixed-Pruning* dataflow accelerator. Four *Flexible-Pruning* accelerators were synthesized, one for each dataset/CNN.

Our case study is based on typical smart video surveillance systems that have to deal with numerous cameras (IoT devices) sending inference requests (frames) to a local Edge server. Therefore, we have set 20 IoT devices to produce inference requests at the real-time rate of 30 FPS. Evaluations are 25 seconds long. Due to factors like FPS fluctuation (SHUAI et al., 2020), network congestion (FU et al., 2014), or variable number of connected nodes, the rate of incoming inference requests (workload's incoming FPS) in an inference server changes over time (REDDI et al., 2020). Based on such environments (SHUAI et al., 2020; FU et al., 2014), we evaluate AdaFlow under two scenarios: **Scenario 1** that represents a more stable Edge environment with 30% random workload deviation every 5 seconds; and **Scenario 2** that represents a more unpredictable environment with 70% workload deviation every 500ms. Experiments are executed 100 times, and average values are reported. We have set the maximum accuracy loss (Accuracy Threshold in Figure 4.2) to 10%. The Runtime Manager selects accelerator type given a predefined criterion ($AccelCriteria$ in Algorithm 4.2). Based on our experiments, we set this value to 10× the reconfiguration time. We note that both parameters are user-defined and can be tuned to specific applications and user goals. In the next section, we evaluate AdaFlow over the Original FINN (baseline) on performance, power efficiency, and QoE[2].

4.2.2.2 Evaluation

Figure 4.4 – (a) FPGA Resource for FINN, Flexible and Fixed accelerators. Accuracy vs. Energy for CNV2W2A on CIFAR-10 (b) and GTSRB (c).



AdaFlow's Design Space. Before we evaluate AdaFlow under the application, this section shows the design space enabled by AdaFlow's Library. Figure 4.4(a) shows the FPGA resource usage (y-axis) for the original FINN and AdaFlow's Flexible and Fixed-Pruning accelerators for CNVW2A2 CNN on the CIFAR-10 dataset. Figures 4.4(b) and (c) plot the energy per inference (x-axes) vs. accuracy (y-axes) for CNVW2A2 on CIFAR-10 and GTSRB datasets. CNVW1A2 follows the same behavior. As it can be noticed in Figure 4.4(a), the Flexible-Pruning accelerator presents the highest resource usage compared to FINN and its Fixed-Pruning counterparts as it requires extra logic to implement the runtime-controllable behavior. Still, as the space taken by feature maps and weights only decreases with pruned models when compared to FINN, Flexible-Pruning shows no increase in BRAM usage (which is often the limiting factor for FPGA-based

[2]QoE is the product of accuracy per the rate of processed inferences.

Table 4.2 – Frame Loss, QoE, Power, and Power Efficiency for all datasets and CNN models on the full 25 seconds run.

		Frame Loss (%)		QoE (%)		Power (W)		Power Eff.
Dataset / Model	Scen.	AdaFlow	Orig. FINN	AdaFlow	Orig. FINN	AdaFlow	Orig. FINN	w.r.t FINN
CIFAR-10 / CNVW2A2	1	0	23.00	81.74	68.32	1.01	1.07	1.39x
	2	5.11	30.99	78.54	61.23	1.20	1.07	1.25x
GTSRB / CNVW2A2	1	0	23.53	65.12	53.55	1.01	1.07	1.40x
	2	3.64	29.91	63.21	49.08	1.14	1.07	1.30x
CIFAR-10 / CNVW1A2	1	12.27	23.68	73.58	66.63	0.98	1.00	1.17x
	2	21.89	31.73	66.12	60.47	1.12	1.00	1.01x
GTSRB / CNVW1A2	1	0	22.57	65.85	69.86	0.94	0.96	1.35x
	2	4.14	31.36	62.88	47.95	1.11	0.97	1.23x

CNN accelerators - i.e., the resource with the highest usage, see Figure 4.4(a)). Flexible-Pruning increases in 1.92$\times$ the number of used LUTs compared to the original FINN. Fixed-Pruning, on the other hand, presents reductions in LUT usage ranging from 1.5% (at 5% pruning rate) to 46.2% (at 85% pruning rate) compared to the original FINN.

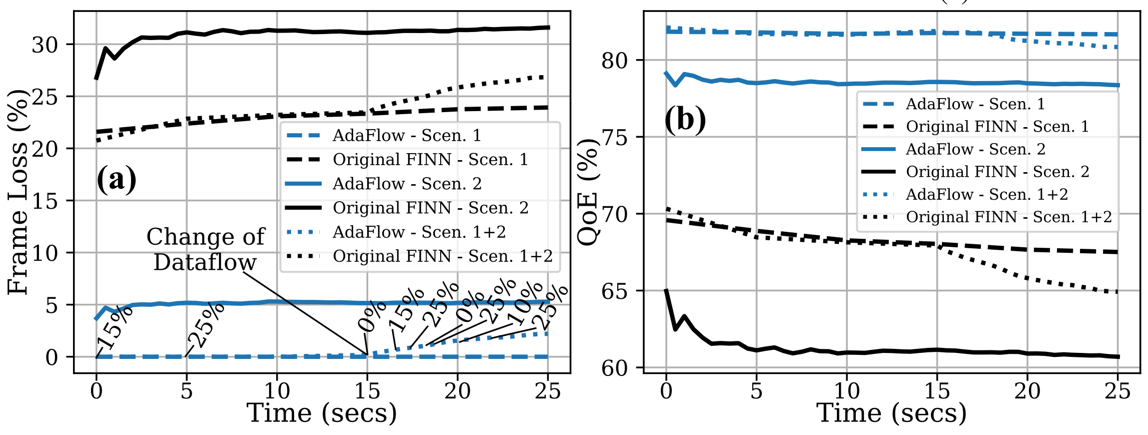
Although requiring more resources, AdaFlow's Flexible-Pruning allows switching to pruned models that will deliver higher performance and lower energy consumption at runtime. Overall, as the pruning rate increases, the energy consumption decreases at the cost of accuracy w.r.t the FINN baseline. For example, it would be possible to switch to a 25% pruned model on Flexible-Pruning (green square in Fig. 4.4(b)), reducing the energy per inference by 1.38$\times$ with accuracy loss of only 9.9% when compared to FINN. The same pruning rate with a Fixed-Pruning accelerator (red square in Fig. 4.4(b)) reduces by 1.64$\times$ the energy consumption w.r.t FINN. When comparing AdaFlow's accelerators, the Fixed-Pruning ones present slightly better inference latency (up to 3.7% difference, 0.67% average), meaning that Fixed-Prune's lower energy consumption is mainly due to having no runtime-controllable logic.

Even though Flexible accelerators present energy consumption higher than their Fixed counterparts, they require no FPGA reconfigurations to switch models at runtime. Consequently, in scenarios where the application requirements change rapidly, Flexible-Pruning is the only alternative for runtime adaptation, as will be presented next.

AdaFlow at runtime. This section evaluates AdaFlow under the two Edge scenarios presented before. Table 4.2 reports frame loss, QoE, and power for AdaFlow and Original FINN averaged over the full 25 seconds run. Table 4.2's right-most column gives AdaFlow's power efficiency (number of processed inferences per Watt) w.r.t Original FINN. First, we see that AdaFlow achieves greater performance (i.e., lower frame loss) and power efficiency than the original FINN for all evaluated datasets and CNNs. The higher performance levels (frame loss up to 27.22% lower than Original FINN - GTSRB/CNVW1A2) and efficiency (up to 1.40$\times$ more efficient - GTSRB/CNVW2A2) are enabled by AdaFlow's dynamic adaptation. By switching to models of higher throughput whenever needed, AdaFlow can adapt the inference processing to eventual workload increases with minimal accuracy loss. This minimal accuracy loss and improved performance also cause AdaFlow to deliver QoE higher than FINN. As it is not always necessary to keep accuracy close to the threshold (e.g., for a currently low workload level), AdaFlow's Runtime Manager grants a 7.07% (CIFAR-10/CNVW2A2) maximum accuracy drop over all evaluations (4.6% on average). We also would like to note that for applications that tolerate accuracy thresholds larger than the one in use (10%), larger performance and efficiency gains are expected since, in that case, CNNs of more aggressive pruning would be allowed.

Now, let us consider a representative evaluation to detail the AdaFlow behavior. In Figure 4.5(a), we show FINN and AdaFlow frame losses (y-axis) over the 25 seconds run (x-axis) for CIFAR-10/CNVW2A2 under Scenarios 1 and 2 as well as Scenario 1+2 that provides an additional evaluation. To show how AdaFlow adapts to a change of environment, Scenario 1+2 starts with a stable condition up to 15 seconds (same setting of Scenario 1) and then shifts to a more unpredictable phase (same setting of Scenario 2) that lasts until the end of the evaluation. In Figure 4.5(a), we also show the pruned models used by AdaFlow for Scenario 1+2 and the moment it changes the accelerator type. Pruned models switched during the execution of other scenarios are not displayed for the sake of clarity. Figure 4.5(b) follows the same idea of (a) for showing Original FINN and Adaflow QoE curves.

Figure 4.5 – (a) Frame Loss, lower the better, and (b) QoE, higher the better, for CNVW2A2 on the CIFAR10 dataset. AdaFlow's model switches are indicated in (a).



As explained earlier, the Runtime Manager selects Fixed-Pruning accelerators for stable environments (i.e., the ones that cause relatively few model switches). Scenario 1 represents such a condition, allowing that AdaFlow reconfigures the FPGA with Fixed-Pruning accelerators. For example, considering the first run for CIFAR-10/CNVW2A2 out of the 100 runs averaged ("AdaFlow - Scen. 1" in Figure 4.5), AdaFlow switched models five times, resulting in a total of five FPGA reconfigurations (around 725ms total). Even though they need FPGA reconfigurations, Fixed-Pruning accelerators offer higher power efficiency: see AdaFlow's lower frame loss and lower power dissipation in Scenario 1 (when using Fixed-Pruning) in contrast to AdaFlow in Scenario 2 in Figure 4.5(a). In Scenario 2, when changes in workload happen at a higher rate, FPGA reconfigurations are not allowed. Thus, AdaFlow employs the Flexible-Pruning accelerator. A total of 31 model switches were performed over Scenario 2's 25 seconds run. For those model switches, however, no FPGA reconfiguration is required thanks to the Flexible-Pruning accelerator. In summary, enabling fast model switches on the FPGA-based server made it possible to achieve power efficiency 1.25$\times$ greater and frame loss 25% lower than the Original FINN in Scenario 2.

Unlike the first two scenarios, Scenario 1+2 presents a shift in the workload condition at runtime (as explained earlier). This change of workload condition causes AdaFlow to change from Fixed- to Flexible-Pruning accelerator. During the more stable phase of Scenario 1+2 (from 0 to 15secs), AdaFlow switched models twice (15 and 20% pruning rates, indicated in Figure 4.5(a)) with Fixed-Pruning accelerators. After that, the Runtime Manager flagged a change in workload at a shorter time interval and reconfigured the FPGA with the Flexible-Pruning accelerator ("Change of Dataflow" in Figure 4.5(a)). Once the Flexible-Pruning was loaded, six more model switches were performed with no need for FPGA reconfigurations. Overall, in Scenario 1+2, AdaFlow achieved a frame loss 24% lower, and increased QoE and power efficiency by 15% and 1.21$\times$, respectively, over FINN. Therefore, in contrast to inference on traditional dataflow accelerators, AdaFlow adapts the inference processing by changing the pruning rate at runtime. Moreover, as shown in Scenario 1+2, AdaFlow can exploit power-efficient accelerators (e.g., from 0 to 15secs) or fast model switching (e.g., from 15secs onwards) dynamically, depending on the workload conditions.

4.3 Workfront 2: Pruning and Early-Exit

Pruning and Early-Exit have already been independently and successfully employed in the state-of-the-art, but no work has shown the benefits of combining these two optimizations (recall Table 3.6). Combining these strategies is non trivial since, traditionally, pruning is defined and applied before the implemented CNN model is deployed (i.e., retraining at design time) while early-exit is a dynamic technique in nature (i.e., works during the inference). In this context, this workfront approaches the following challenges: (*i*) enabling their simultaneous use *at runtime* to adapt the inference processing; and (*ii*) in considering their combined impact on accuracy and inference costs. We called this workfront AdaPEx for Adaptive Pruning of Early-Exit CNNs (KOROL et al., 2023b).

4.3.1 AdaPEx

Figure 4.6 – Workflow for the AdaPEx workfront.

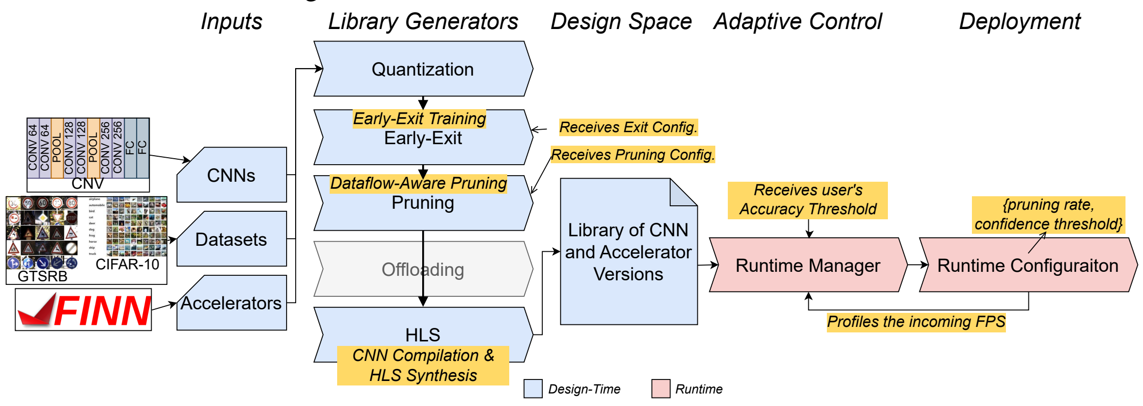


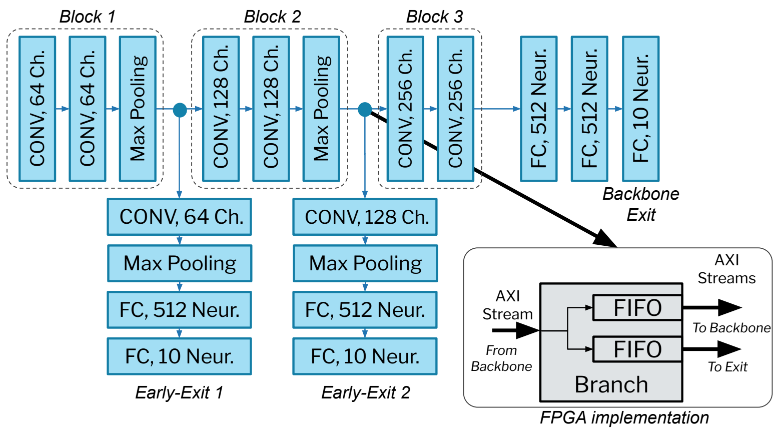
Figure 4.6 details the AdaPEx workflow. AdaPEx specializes the base workflow from Figure 2.8 with the quantization, pruning, early-exit, and HLS optimizations. In the Input step at design-time, AdaPEx receives the FINN accelerator code with the CNV description and the datasets (same inputs as the previous workfront). AdaPEx is evaluated with CNV quantized at two bits for activation and weights (i.e., CNVW2A2). The next steps take place to automatically generate the library with pruned early-exit models with different resource and accuracy profiles. Particularly, AdaPEx first adds and trains the early-exits (Early-Exit Training, where it accepts a configuration file specifying the early exits, Exit Config. in Figure 4.6), then the same Dataflow-Aware Pruning and CNN compilation & HLS Synthesis optimizations from last workfront (Section 4.2) with some small modifications to treat the exits (presented below) are used to prune and synthesize the CNNs. This is done by varying the pruning rate at fixed steps (like in previous workfronts), gathering multiple pruned versions of each early-exit CNN.

At runtime, AdaPEx's Runtime Manager carries out the Adaptive Control by dynamically choosing the best pruning rate (as in the previous workfront) and confidence threshold (regarding the early-exit, see the early-exit background in Section 2.1.3.1) to adapt the inference processing, automatically reconfiguring the FPGA as needed. Next, we present the AdaPEx's early-exit and the modifications made in this workfront to the pruning and HLS optimizations in the Library Generators step and the Runtime Manager for the Adaptive Control step.

4.3.1.1 Library Generators

Early-Exit Optimization. A series of modifications to the existing FINN/Brevitas design flow were required to enable inference on CNN models with multiple branches. Starting from a regular Brevitas CNN model, AdaPEx can attach the early-exits to any location along the CNN topology. The user can specify how AdaPEx adds the early exits (through the early-exit configuration file), setting the location and the operations (e.g., CONV, FC, etc.) of the exits, which also enables easy exploration of the design space.

Figure 4.7 – The CNV with two early exits used in our experiments.



Let us take as example the CNN used as case study in this work, the CNV (see Subsecion 4.1), a VGG-like CNN available in FINN. We set AdaPEx to add two early exits to CNV (Figure 4.7): one after the second CONV layer (first block) and another exit after the fourth CONV layer (second block). As for configuring the early exits, we set AdaPEx to append a CONV layer (with the same configuration of the block, number of channels, filter size, etc.) followed by a max pooling layer with kernel size of $k = \lfloor \frac{DIM}{2} \rfloor$, where DIM is the dimension of the block's output feature map, so it significantly reduces the map size, making FPGA synthesis feasible for the two following FC layers (that use the same configuration as the FC layers in the original CNV - FC with 512 and

10 neurons in Figure 4.7). This exit configuration used here as case-study (with CONV, Max-Pool, and FC layers) follows previous works on early-exit (TEERAPITTAYANON; MCDANEL; KUNG, 2016; PANDA; SENGUPTA; ROY, 2016).

After the model is fully defined with all its exits in Brevitas, the model can be trained. AdaPEx uses a training procedure (TEERAPITTAYANON; MCDANEL; KUNG, 2016) implemented as a Brevitas script where all exits are simultaneously trained (Joint Loss Function, see Equation 2.7). Also, AdaPEx takes the softmax on each exit as a measure of their confidence.

From the FPGA point of view, a new HLS module had to be developed and included in the FINN design flow, along with a new transformation step to manage it during FPGA mapping. This new HLS *branch* module performs the branches between the CNN backbone and the early exits (*FPGA Implementation* in Figure 4.7). FINN connects MVTU modules (the HLS module in charge of executing each CONV and FC layer) with AXI stream interfaces (implemented as FIFOs, in the input/output ports of each MVTU). In that way, the output of every HLS module corresponds to the CNN layer's output feature map and is fed to the next module in the dataflow. The new branch module leverages this implementation style by duplicating the incoming stream (AXI Stream from backbone in Figure 4.7) into two independent AXI Streams (one feeding the backbone, the other feeding the early exit). Therefore, some FPGA resource overhead will be observed (mainly in terms of BRAMs), but neither backbone nor exit throughput is undermined, and there is no risk of pipeline stalls. More implementation details on how we add the early exits (on both hardware and software implementations) are given in Appendices F and G.

Pruning Optimization. To prune out parts of a CNN that will be later executed on the FPGA, this workfront extends the Dataflow-Aware Pruning from the previous workfront in Section 4.2. In this case, two approaches can be taken when pruning an early-exit model: pruning only the backbone layers or pruning the backbone and the CONV layers inside each early-exit. While the latter approach will speed up the early exits as well as the backbone layers, it may significantly decrease the exit's accuracy since these exits can be less resilient to pruning. On the other hand, the former approach, which leaves the exits CONV layers untouched, offers an exciting trade-off between performance and accuracy when comparing the early to the last exits. AdaPEx supports both approaches (which the user can set to either one or both with a *pruned* flag in the early-exits configuration). These approaches will be compared in the following results section. After pruning the early-exit CNN, it is retrained and exported as an ONNX file to be compiled by FINN (the same as the Fixed-Pruning accelerators from the previous workfront, the Flexible-Pruning accelerators that do not require FPGA reconfiguration were not used in this workfront).

4.3.1.2 Adaptive Control

Runtime Manager. AdaPEx's Runtime Manager follows the Algorithm 2 and the same rationale used before: whenever a change in the workload is flagged (possible with performance monitors added to the software in charge of the incoming inferences), the runtime manager searches in the library for a new configuration that is most adequate to the current workload and the user's accuracy threshold. Only now, configurations involve pruning rate *and* confidence threshold. Thus, it can either change the confidence threshold, changing the acceptability of the earlier exits (as seen in Section 2.1.3.1, lowering the threshold cause more inputs to get classified earlier); or, change the pruning rate. We note that changing the confidence threshold does not require any change in the accelerator and, thus, no FPGA reconfiguration. This is due to the fact that in our accelerators all exits get mapped to the FPGA, and accepting or not an output is not decided on the accelerator.

4.3.2 Results

4.3.2.1 Evaluation Scenario

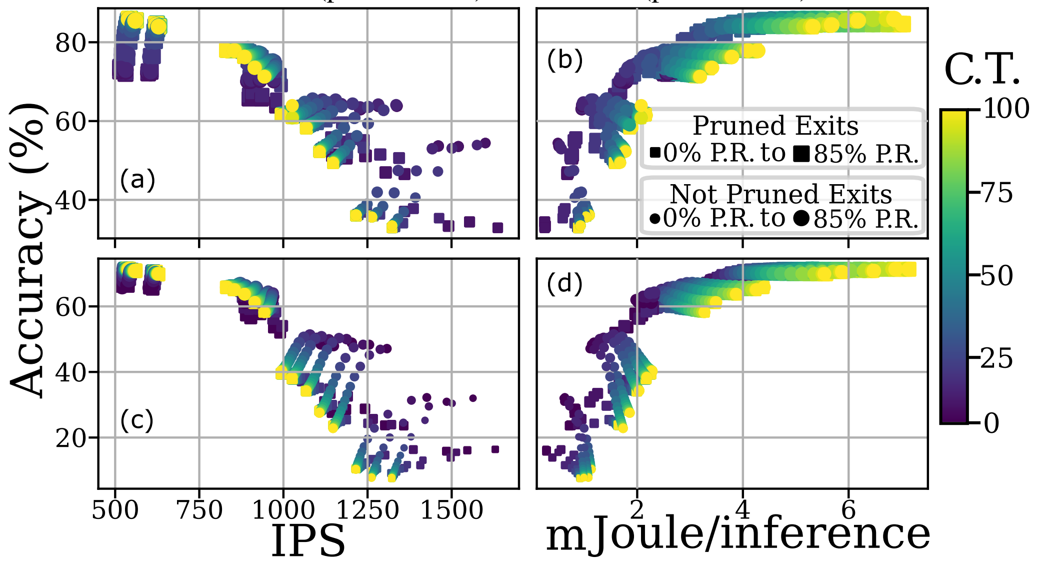
In this workfront, we focus the evaluation on FINN with 2-bits quantization (CNV-W2A2). AdaPEx generates 18 models for each initial early-exit CNN with pruning rates from 0% (not-pruned) to 85% (5% steps). Each model generates a specific FINN accelerator (i.e., only fixed accelerators are used for this evaluation). On each pruned model, the confidence threshold can vary from 0 to 100% at 5% steps. The early-exit training procedure follows (TEERAPITTAYANON; MCDANEL; KUNG, 2016), weighting the first exit at 1.0 and the remaining at 0.3.

We base our evaluation on a typical IoT application, smart video surveillance, with numerous cameras requesting frames to be inferred at a local Edge server. To keep the discussion general, we will refer to those requests as inference requests. For that, we model 20 cameras requesting inferences at the rate of 30 Inferences per Second (IPS) for 25 seconds. Due to factors like IPS fluctuation, network congestion, or the variable number of connected cameras, the rate of incoming inference requests (workload) changes over time (REDDI et al., 2020), represented as 30% random workload deviation every 5 seconds (same as Scenario 1 in Subsection 4.2.2 from previous workfront). Experiments are executed 100 times, and average values are reported. We have set the maximum accuracy loss (AdaPEx's accuracy threshold) to 10%. Next, we evaluate AdaPEx over the following baselines:

- Original **FINN** accelerator synthesized to off-the-shelf CNN models;
- Pruning-Only, called **PR-Only**, that uses the runtime selection, but with a single (no early) exit to evaluate the pruning benefits;
- Confidence-Only, a not-pruned early-exit model, called **CT-Only** that also uses the runtime selection but adapts the Confidence Threshold only.

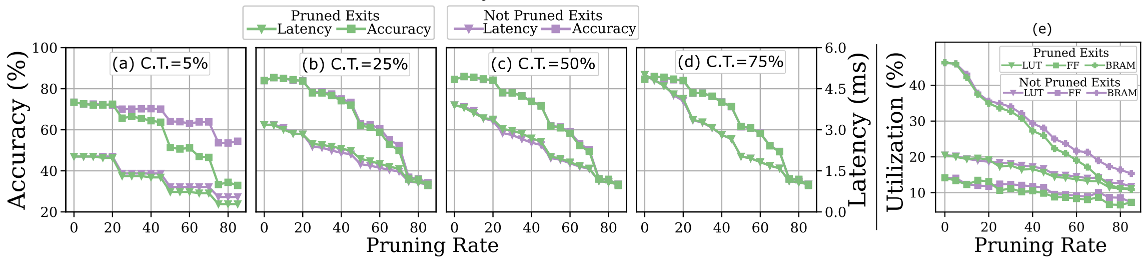
4.3.2.2 Evaluation

Figure 4.8 – AdaPEx design space for CNVW2A2 on IPS and Joule per inference on CIFAR-10 dataset (plots a and b) and GTSRB (plots c and d).



AdaPEx's Newly Created Design Space. Figure 4.8 presents the design space enabled by the AdaPEx combination of pruning and early-exit for the two evaluated datasets: CIFAR-10 (upper plots) and GTSRB (lower plots). Figures (a) and (c) plot throughput (in IPS) versus accuracy (y-axis) while Figures (b) and (d) give the energy per inference versus accuracy. Design points are generated by varying the pruning rate (P.R.) from 0 to 85% (indicated by the size of each point) and the confidence threshold (C.T.) from 0 to 100% (indicated by the color scale) for both pruned exits (squares) and

Figure 4.9 – Accuracy (left y-axes) and latency (right y-axes) over pruned CNNs with Confidence Thresholds (C.T.) of 5, 25, 50, and 75% on plots from (a) to (d). On plot (e), resource usage on a XCZU7EV board for the Early-Exit CNVW2WA on the CIFAR-10 dataset.



not pruned exits (circles). From plots (a) and (c), we see that CNN models (and accelerators) of lower accuracy that run faster are required to cope with high workload levels. A similar behavior is observed on the energy plots (b and d), but with a noticeable plateau from around 4mJ onwards for both datasets. Beyond 4mJ, the extra energy consumed by targeting inferences of higher accuracy is wasted. As can be observed, it is crucial to match the CNN model, by setting P.R. or C.T., to the current workload at runtime. This way, no accuracy is unnecessarily lost while meeting throughput constraints with minimal energy.

Pruned Early-Exits: One important design decision of pruning early-exit CNNs is how to handle the exit layers, i.e., whether or not to prune layers in the early exits. As shown in Figure 4.8, having these two options naturally enlarges the design space, but what is more interesting is that it may also recover some of the accuracy lost from pruning the original CNN layers (backbone). Figures 4.9(a)–(d) present the average accuracy (left y-axes) and latency (right y-axes) versus pruning rate (x-axes) on the CIFAR-10 dataset under four Confidence Thresholds (C.T. = 5%, 25%, 50% and 75%). Figure 4.9(e) plots the FPGA resource utilization. In *Pruned Exits* across all plots, the additional convolutional layers for the early-exits are pruned at the same rate of the backbone. In *Not Pruned Exits*, the exits are not pruned when added to the CNN backbone.

In terms of accuracy, not pruning early exits recovers some of the accuracy for the more heavily pruned models especially at lower confidence thresholds (5 and 25% thresholds in plots (a) and (b)). The reason for this is twofold. First, note that the layers in the pruned backbone are larger than the not pruned exit layers. As the backbone layers get more heavily pruned, their accuracy drops quicker than that of the early exits. Second, the confidence threshold also plays an important role in such scenarios. As the confidence threshold is lowered, more inputs get classified at earlier exits, increasing their impact in the overall accuracy/latency when the full test set is considered. When considering latency, similar reasoning holds for combinations of high pruning rates and low confidence thresholds. For example, from around 50% pruning onwards in plot (a), where the low confidence threshold causes more inputs to exit early, we see faster inferences.

In summary, in scenarios where a high pruning rate is needed to, for example, achieve high throughput, not pruning early exits may help recover some of the accuracy lost in the backbone. In high-accuracy scenarios, in turn, lightly pruned CNNs can be used with a high confidence threshold, causing the last (backbone) exit to be in charge of classifying most inputs. In this case, early exits can be pruned more aggressively, reducing resource usage without significant costs in accuracy.

FPGA Resource Utilization: Figure 4.9(e) plots the BRAM, LUT, and FF resource usage w.r.t. pruning rates for "Pruned" and "Not Pruned" early-exit CNNs. Remember that the confidence threshold is simply used to accept output vectors and thus it does not change any hardware configuration. For this reason, the resource plot is valid for all confidence thresholds. As can be seen, there is no significant difference in resource usage when comparing pruned and not-pruned early exits for lightly pruned CNNs (from 0 up to 20% pruning rates). This is due to the reduced contribution of early-exits to the total resource usage of such large models. For example, for the not-pruned early-exit CNN (0% pruning rate), exits correspond to 15.25%, 22.58%, and 30% of the allocated BRAMs, LUTs, and FFs, respectively. On the other hand, for the CNN pruned at 85% the exits represent 45%, 28.38%, and 30.82% of the accelerator BRAM, LUT, and FFs. Meaning that as the pruning rate increases, the exits impact on the total resource usage rises and the cost of the "Not Pruned" (purple curves) exits become clearer in contrast with their pruned counterparts (green curves). This is most noticeable for the BRAM usage as this type of resource is primarily used to implement FIFOs to store intermediate data (i.e., feature maps), which are larger within the not pruned exits.

We see that combining pruning and early-exit leads to highly heterogeneous design space. From Figures 4.8 and 4.9, there are many different combinations of pruning rate and confidence threshold, resulting in different accuracy, performance, and energy profiles. The challenge is how to tap into this potential with an effective dynamic approach at runtime. In the following section, we evaluate how AdaPEx achieves this.

AdaPEx's Runtime. Table 4.3 summarizes the evaluation on the CIFAR-10 and GTSRB datasets. It presents the rate of lost inference requests (Infer. Loss), accuracy, and latency results averaged over all executions (25s each). Results show that AdaPEx delivers the best performance across both datasets, reporting no inference loss, meaning

Table 4.3 – Averaged Inference Loss, Accuracy, and Latency over the full 25 seconds run.

	Dataset	Infer. Loss[%]	Accuracy[%]	Latency[ms]
AdaPEx	CIFAR-10	0.00	80.15	3.52
	GTSRB	0.00	68.80	3.04
PR-Only	CIFAR-10	11.82	85.72	4.37
	GTSRB	0.00	65.38	3.79
CT-Only	CIFAR-10	12.58	86.57	4.38
	GTSRB	14.01	66.09	3.63
FINN	CIFAR-10	22.80	88.74	5.19
	GTSRB	23.60	70.04	5.21

1.31× and 1.32× increase in the number of processed inferences over the original FINN accelerator on CIFAR-10 and GTSRB datasets, respectively. AdaPEx also processes inference requests at latency 1.48× and 1.72× lower than FINN on CIFAR-10 and GTSRB. The increased performance comes at a moderate accuracy cost. AdaPEx processed inferences with accuracy 8.59% below the original CNN (running on FINN) on CIFAR-10 and only 1.24% below on the GTSRB dataset. Recall that this cost is controlled by the user through the accuracy threshold, which was set to 10% in our evaluations. As discussed next, however, by smartly selecting pruning rates and confidence thresholds, AdaPEx is able to deliver inferences of higher quality of experience and energy efficiency.

Figure 4.10 – Average EDP normalized w.r.t original FINN accelerator (bars) and QoE (curves) for CIFAR-10 and GTSRB datasets.

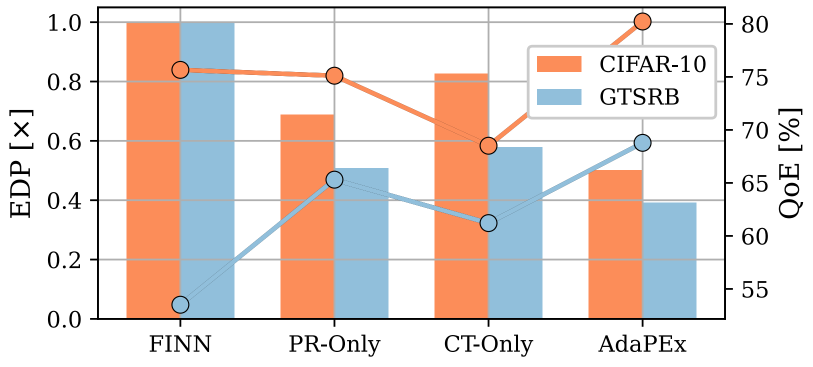


Figure 4.10 plots the Quality of Experience (QoE curves) and the averaged Energy-Delay Product (EDP) w.r.t original FINN accelerator (bars) averaged over all executions. With QoE (defined as the product of accuracy by the percentage of processed frames) we can assess the overall inference serving quality by measuring the performance-accuracy trade-off. AdaPEx achieves the highest QoE levels among all baselines, increasing QoE over FINN by 11.72% and 15.27% on the CIFAR-10 and GTSRB datasets, respectively. This is because the accuracy loss due to pruning is in part compensated by tuning the confidence threshold.

For example, considering the first run on the GTSRB dataset, AdaPEx changed the pruning rate four times (between 5, 20, and 30% pruning rates, requiring four FPGA reconfigurations that took 580ms in total). With these pruning rates, four confidence thresholds (30, 40, 55, and 60%) were used. Such trade-off can only be exploited by AdaPEx since it simultaneously searches for the best match between pruning rate and confidence threshold.

Regarding energy efficiency, AdaPEx shows also a highly positive impact. Figure 4.10 bars show that AdaPEx reduces the average EDP in 2$\times$ on CIFAR-10 and 2.55$\times$ on GTSRB w.r.t original FINN accelerator. By selecting pruned early-exit CNNs, AdaPEx gets the best from both optimizations, resulting in a overall efficiency higher than the baselines pruning only (PR-Only that cannot exploit easy images to lower the latency) or early exiting only (CT-Only that cannot alleviate some the power cost of large models).

4.4 Workfront 3: Pruning and Early-Exit on Binary-decided Offloading

This is the first of two workfronts considering offloading in this thesis. In the first one, the design space created by adding offloading on top of pruning and early-exit is explored statically. It investigates when (i.e., according to some optimization goal like latency, power, or throughput) it is worth to execute inferences locally on a smaller but optimized IoT (that do not need to send data over the network) *or* offload inputs to a more powerful edge server. Thus, treating offloading as a binary decision. This workfront will also show how variations in the environment (fluctuations in the IoT-edge connection) can affect this decision, and how other optimizations like pruning and early-exit can alleviate some of the computational burden, helping to recover some of the time lost in the IoT-edge communication. This workfront presents the ExpOL framework for Exploring the design space of Offloaded and Local FPGA-based inference processing (KOROL et al., 2023a).

4.4.1 ExpOL

Figure 4.11 – Workflow for the ExpOL workfront.

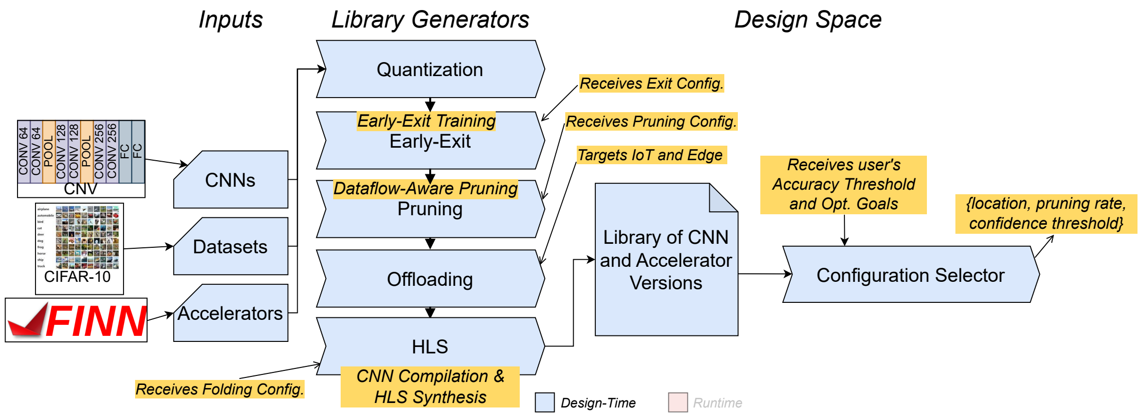


Figure 4.11 details the ExpOL workflow. Regarding the general workflow from the beginning of the chapter, Figure 4.1, ExpOL has the following particularities:

- Most importantly, ExpOL has no runtime steps. Only the design-step optimizations are used in this workfront. Thus, runtime steps are replaced by a *Configuration Selector* that performs a one-time search in the Library;
- Under the Library Generators step, the offloading optimizations regards the generation of CNNs and accelerators to be deployed on two devices, IoT and edge server;
- For the evaluated inputs, CNV on CIFAR-10 dateset is used with FINN accelerators.

As a design-time only workfront, ExpOL works before deployment to output a library of tuned design *configurations* generated based on a set of user goals. An ExpOL configuration specifies the location of deployment (IoT device or edge) with the corresponding FPGA bitstream and CNN model (with defined pruning and early-exit parameters).

Besides the CNN, datasets, and accelerator, ExpOL also receives folding and early-exit configuration files, an accuracy threshold, and the user optimization goals. Optimization goals are specified as a tuple of α, β, and γ weights for combining the designs accuracy, performance, and power (Opt. Goals in Figure 4.11). The quantized CNNs are, first, sent to the Early-Exit Training before they get pruned by the Dataflow-Aware Pruning step, creating multiple versions of the original CNNs. Then, these CNNs go through the Offloading step that, in ExpOL, is responsible for calling the CNN Compilation & HLS Synthesis for the two devices: IoT and edge boards. Those versions are then input to the HLS step to build the design space (same procedure of the previous workfront). Finally, the Configuration Selection searches for the best-scoring configurations according to the user optimization goals and accuracy threshold.

4.4.1.1 Library Generators

Quantization, Early-Exit, and Pruning. The ExpOL workfront follows the same procedure for *quantizing*, adding *early-exits* and *pruning* the user's CNNs as the previous workfronts (recall Section 4.3 and 4.2). For the HLS step, however, differently from the previous workfronts that target a single device, ExpOL utilizes different folding configuration files and multiple target FPGA devices.

Offloading and HLS. In FINN, the user can tune the parallelism of every HLS module through a JSON folding configuration file (*Folding Config.* in Figure 4.11). By increasing the parallelism in the folding configuration, it is possible to increase the accelerator throughput without changing the CNN models, trading FPGA resources (and power) per performance. The Offloading optimization will simply define two FPGA devices (with their folding configuration file) as the two possible locations for inference processing: In the ExpOL case study, the folding used for the IoT FPGA is $4\times$ narrower than the one for the edge server (that uses the FINN default folding for CNV, recall Section 2.2.2.2 on FINN folding configurations). As will be shown in the Evaluation Section below, this lower parallelization already requires almost all slices of the FPGA on the IoT Device. Lastly, these configurations will be used in the HLS step for generating the bitstream files as well as the reports on power dissipation (from Vivado synthesis) and performance (from RTL simulations) to support the configuration search.

4.4.1.2 Configuration Selector

The Configuration Selector performs the last design-step (Figure 4.11). It will exhaustively navigate the design space looking for the best configuration. The search is based on a profit equation configured by the user with three weights for combining accuracy (α), throughput (β), and power (γ) as

$$\begin{gathered}\text{profit}_i = \alpha \cdot \text{config}^i_{\text{accuracy}} + \\ \beta \cdot \text{config}^i_{\text{throughput}} + \gamma \cdot (1 - \text{config}^i_{\text{power}})\end{gathered} \tag{4.1}$$

where $\text{config}^i_{\text{throughput}}$, $\text{config}^i_{\text{power}}$, and $\text{config}^i_{\text{accuracy}}$ are the i-th configuration's throughput, power dissipation, and accuracy normalized w.r.t all configurations in the design space (min-max normalization).

The search starts by filtering out all design points with accuracy below the minimum specified by the user (*Accuracy Threshold* in Figure 4.6). After that, ExpOL evaluates the profit equation on all design points left. These design points are then sorted, so the Configuration Selector outputs the configuration of the highest profit. In case the user inputs more than one tuple, a library of optimal design points is generated by outputting one configuration for each tuple. These configurations represent the *Pareto front* for the specified optimization goal and are ExpOL main product. It is the user responsibility to make the best use of the ExpOL library. For example, ExpOL can enable different "operating modes". Assuming a battery-powered IoT device, the ExpOL library can be used to switch to a low-power configuration (e.g., one generated for a tuple with a high γ value) when the battery level approaches a critical value.

4.4.2 Results

4.4.2.1 Evaluation Scenario

In this workfront, we adopted the CNV CNN from FINN with 2-bit quantization (CNVW2A2) on the CIFAR-10 dataset (3x32x32 images). ExpOL generates 18 models for the early-exit CNN with pruning rates from 0% (not-pruned) to 85% (5% steps). Each pruned CNN's confidence threshold vary from 0 to 100% at 5% steps. Synthesis target a PYNQZ1 for the IoT device and a ZCU104 for the edge server.

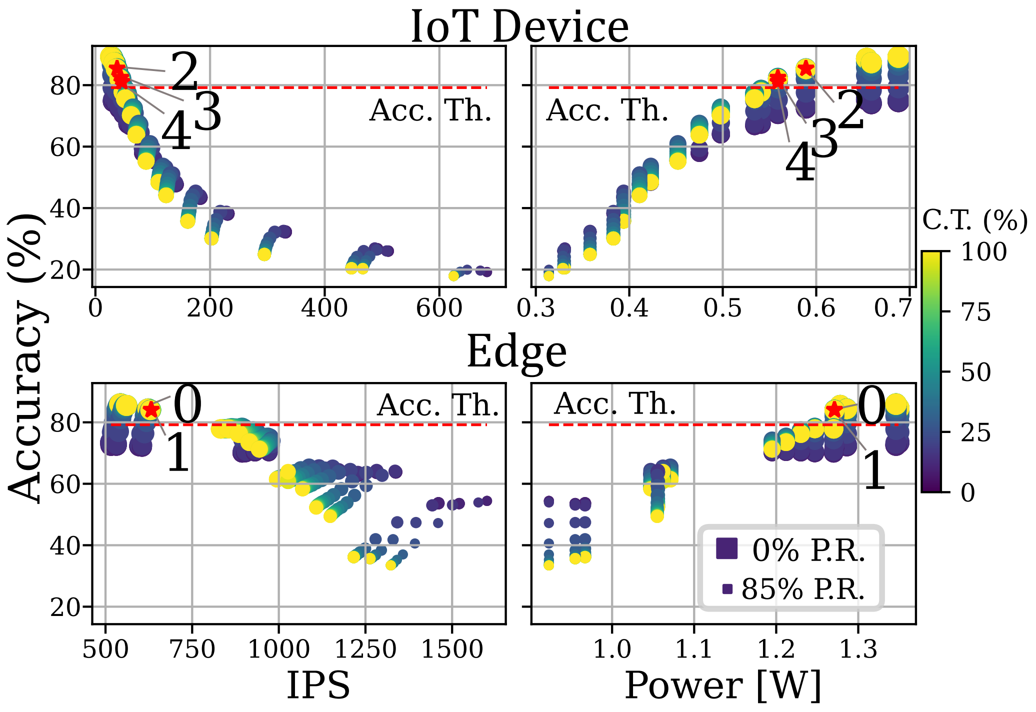
The workfront is evaluated on an IoT smart video application that can *request inferences to an edge server or process it locally*. Evaluations are 15 seconds long. The IoT device produces 30, 60, and 90 Inferences per Second (IPS) during the first 5, 10, and 15 seconds, respectively. The 4G/LTE Bandwidth Logs dataset (HOOFT et al., 2016) is used to model the communication channel. It provides the measured quality of 4G/LTE connections recorded along different routes in a city while downloading a large file over HTTP. Two traces were chosen for evaluation, representing two scenarios: a *stable* one recorded on a walking person and a *unstable* one recorded on a moving tram.

Table 4.4 – Optimizations tuples and their generated configurations. Opt. Tuple is the $\{\alpha, \beta, \gamma\}$ weights for accuracy, throughput, and power, respectively. P.R. and C.T. give the selected pruning rate and confidence threshold, respectively.

Opt. Tuple	Tuple Description	Name	ExpOL Configurations		
{0.5,0.5,0.0}	*Accuracy and Throughput*	Cfg. 0	Edge	20% P.R.	60% C.T.
{0.0,1.0,0.0}	*Throughput*	Cfg. 1	Edge	20% P.R.	65% C.T.
{0.5,0.0,0.5}	*Accuracy and Power*	Cfg. 2	IoT Device	15% PR.	50% C.T.
{0.0,0.0,1.0}	*Power*	Cfg. 3	IoT Device	20% P.R.	30% C.T.
{0.3,0.3,0.3}	*All important*	Cfg. 4	IoT Device	20% P.R.	45% C.T.

For the following evaluation, we have two baselines: **Original-IoT Device**, the original CNV running locally on the IoT FPGA; and, **Original-Edge** when all inferences are offloaded to the edge FPGA running the original CNV. The user can set any combination of search parameters in ExpOL tuples. However, here we illustrate its capability with five combinations presented in Table 4.4 (with descriptions and their generated configurations). The accuracy threshold to search configurations is set to 10%.

Figure 4.12 – Inferences per Second or IPS (left plots) and Power (right plots) for the IoT device (upper plots) and edge (bottom plots) on the CIFAR-10 dataset with CNVW2A2 model (note the different x-axis ranges).



4.4.2.2 Evaluation

ExpOL Design Space. Figure 4.12 presents the accuracy *versus* Inferences per Second (IPS) and *versus* Power for the FPGAs at the IoT device (a PYNQZ1 board, upper plots) and at the edge server (a ZCU104 board, bottom plots). The size and color of the design points represent their pruning rate and confidence threshold, respectively. We start by highlighting performance and efficiency differences between the two platforms. The larger edge server FGPA delivers the highest throughput levels, achieving more than 1500 IPS (at the highest pruning rate and the lowest confidence threshold). The IoT device, in turn, achieves a maximum of 683 IPS. However, the lower throughput delivered by the smaller IoT FPGA comes with a significantly lower power dissipation, and a higher power efficiency for most design points.

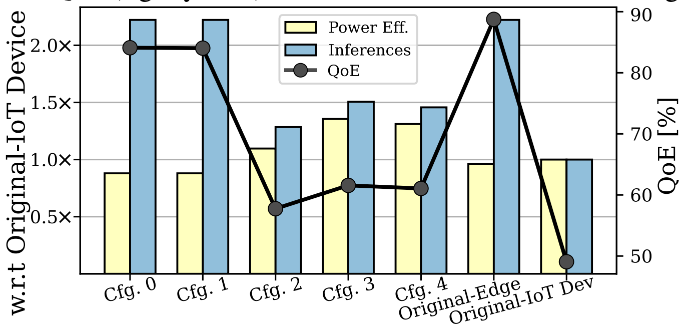
Table 4.5 – FPGA resource usage of selected configurations and baselines.

Configuration	FPGA	LUTs	FFs	BRAMs
Cfg. 0	Edge (XCZU7EV)	42898 (18.62%)	54167 (11.75%)	111 (35.58%)
Cfg. 1	Edge (XCZU7EV)	42898 (18.62%)	54167 (11.75%)	111 (35.58%)
Cfg. 2	Device (XC7Z020)	30135 (56.64%)	35990 (33.83%)	98 (70.36%)
Cfg. 3	Device (XC7Z020)	29106 (54.71%)	34842 (32.75%)	93 (66.79%)
Cfg. 4	Device (XC7Z020)	29106 (54.71%)	34842 (32.75%)	93 (66.79%)
Original-IoT Device	Device (XC7Z020)	26051 (48.97%)	30730 (28.88%)	140 (83.93%)
Original-Edge	Edge (XCZU7EV)	31807 (13.81%)	39029 (08.47%)	102 (32.69%)

Table 4.5 shows the resource usage for the baselines and ExpOL generated configurations. For instance, when comparing two configurations with an equal pruning rate of 20% (configurations 0, running at the edge and 3 in the IoT), we see that the larger parallelization at the edge consumes almost 50% more LUTs and 55% more FFs than the same pruning rate on the IoT. Due to small FPGAs at IoT devices, accelerators quickly exhaust their resources. For example, the largest accelerator in the design space (not presented in Table 4.5) occupies 96.07% of the available BRAMs and over 97% of the slices in the IoT FPGA. Such a design leaves no room for more parallelism. Also noticeable is the overhead of the early-exit layers in terms of resource usage. If we compare the Original-Edge to ExpOL configuration 0, we see an increase of LUT (34.86%) and FF (38.78%) utilization due to early-exits - despite configuration 0 being synthesized for a 20% pruned model. In summary, ExpOL creates a design space that ranges from fast and power-consuming to slower but more efficient inference processing. In this space, ExpOL can generate configurations covering a wide range of optimization targets.

ExpOL configurations. Figure 4.12 also shows the five configurations generated by ExpOL according to the optimization tuples from Table 4.4. The horizontal line (*Acc. Th.*) gives the 10% accuracy threshold used throughout our evaluation. By setting the weights of the optimization tuple, the user can trade-off accuracy for performance and power. For instance, tuples that consider performance (e.g., tuples 0 and 1) get allocated at the edge since its larger FPGA can deliver higher throughput levels. In contrast, for tuples targeting power only (tuple 3) or a compromise between power and accuracy (tuple 4), ExpOL deploys solutions that processes inferences locally. From Figure 4.12, we note that the user could increase gains by allowing a lower accuracy threshold, as a lower line would enable more design points to be selected. In summary, by tuning the optimization parameters in ExpOL, the user can tune the design according to any goal, allowing it to match the inference processing characteristics to the application demands.

Figure 4.13 – Power Efficiency and Total of Processed inferences w.r.t Original-IoT Device (bars, left y-axis) and QoE (right y-axis) under stable network scenario. Higher is better.

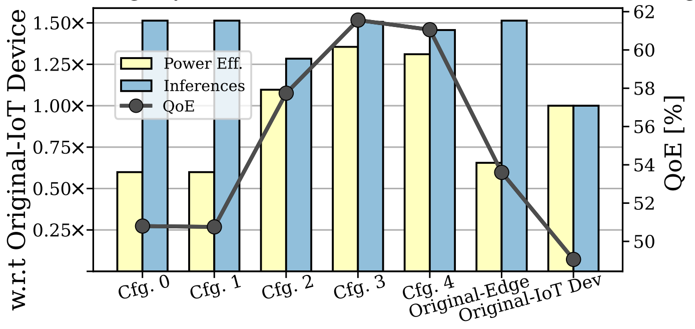


ExpOL at Runtime. We now evaluate the configurations generated by ExpOL under the two IoT-Edge application scenarios from Section 4.4.2.1. First, the *stable* scenario where the available bandwidth (49.2 Mbps on average) has fewer variations and stays at all times above the minimum required to offload the inferences (49.1 Mbps at the highest load). The configurations generated by ExpOL are compared to the baselines in Figure 4.13 on the Power Efficiency (inferences per Watt) and the total of Processed Inferences. Figure 4.13 also presents results on Quality of Experience (QoE, right y-axis). Each configuration (ExpOL and baselines) in Figure 4.13 represents one whole execution under the same scenario. Power Efficiency and Processed Inferences (bars) are normalized w.r.t Original-IoT Device baseline.

In this scenario, we notice a significant performance difference between the edge (Original-Edge baseline and ExpOL configurations 0 and 1) and IoT device (Original-IoT Device and configurations 2-4). Since the network can accommodate all frames offloaded from the IoT device, it does not harm the final throughput and all inferences can be fed to the edge. Also, because of such high bandwidth and the accelerators deployed by the edge configurations having enough throughput, there is no difference in the number of processed inferences between Original-Edge baseline and ExpOL configurations 0 and 1 in Figure 4.13. The high performance also led to high QoE levels with a slight advantage of Original-Edge over configurations 0 and 1. This is due to the original CNN running on the baseline having accuracy higher than the ones deployed by ExpOL configurations 0 (4.64% accuracy loss) and 1 (4.7% loss). We note that in case of heavier workloads (e.g., from multiple IoT clients) we would see the Original-Edge performance (and QoE) decrease. For such heavier workloads, the throughput can only be improved with the ExpOL optimized models.

Regarding efficiency in Figure 4.13, the power-oriented configuration, ExpOL

Figure 4.14 – Power Efficiency and Total of Processed inferences w.r.t Original-IoT Device (bars, left y-axis) and QoE (right y-axis) under unstable network scenario. Higher is better.



configuration 3, is 1.4× and 1.35× more power-efficient than Original-Edge and Original-IoT Device baselines, respectively. However, the smaller IoT accelerators cannot process the full workload. The Original-IoT Device baseline showed an inference loss of 55% throughout the evaluation. While ExpOL configurations 2, 3, and 4 improved it to 42.2%, 32.2%, and 34.4% of inference losses, respectively. Considering those system configurations running inferences locally, ExpOL overcomes the Original-IoT Device baseline thanks to the pruning and early-exit optimizations. Such examples show that optimizations at the CNN level can alleviate the performance losses for power-oriented goals. As we are going to see next, local and optimized configurations will also be helpful when the network shows a not-so-stable behavior.

Similar to the previous plot, Figure 4.14 shows the results under the *unstable* scenario. The lower average bandwidth of this scenario (6.93 Mbps) represents a more challenging environment for edge offloading. Over the 15 seconds of evaluation, 287 frames were lost, representing 31.89% inference loss for edge-based configurations (Original-Edge baseline and ExpOL configurations 0 and 1). For those configurations, the inference loss is also perceived as a significant drop in the delivered Quality of Experience (QoE). In this scenario, the ExpOL configurations running inferences locally achieve QoE *and* power efficiency levels higher than *both* baselines. It becomes clear that tuning the design decisions of the inference processing is crucial for IoT-edge applications and the pruning and early-exit optimizations help navigate the design space when offloading inferences is not beneficial.

4.5 Workfront 4: Pruning and Early-Exit on Split Inference

In the previous workfront, the offloading decision was treated as a binary decision, meaning that the inference processing was configured to run exclusively at the IoT or exclusively at the edge. This workfront, however, aims at an inference processing that, by *splitting* the CNN into two halves, can run part on the IoT *and* part on the edge (with intermediate data, i.e., feature maps, sent over the network). More importantly, these IoT and edge layers can have different optimization levels since they run on nodes that work under different resource/power/energy requirements. For example, when the initial layers of a CNN are processed locally at the IoT and the remaining layers on the edge server, the IoT layers could be more heavily optimized (quantization, pruning, early-exit) to accommodate the IoT's more restricted resource requirements. As for the layers running on the edge server, more relaxed optimization levels can be used, granting a higher accuracy, given that resources are not as restricted.

However, such runtime adaptation on *some* of the layers creates issues. One challenge involves synchronizing the far-located parts of a CNN (with some layers running at the IoT others running at the server). Another is related to pruning/quantization, since changes in pruning rate and bit-width usually require changes in the CNN model (e.g., retraining). Therefore, if the IoT device changes the pruning rate, for instance, the layers on the edge server would have to be switched as well. In other words, we have, on one hand, CNN optimizations like pruning and early-exit working at runtime to adapt the inference processing to current needs. On the other hand, we need to deploy these optimized models on FPGAs while also considering for offloading between IoT devices and edge server. Thus, to enable IoT devices to adapt at runtime and still offload data to the edge whenever needed, this workfront presents the AOI framework for automatically building Adaptive Offloaded Inferences[3].

[3]In Appendix E we present a study that was initially carried out to evaluate a static FPGA-based split inference (i.e., that does not consider the implementation of this workfront with runtime adaptability).

4.5.1 AOI

Figure 4.15 – Workflow for the AOI workfront.

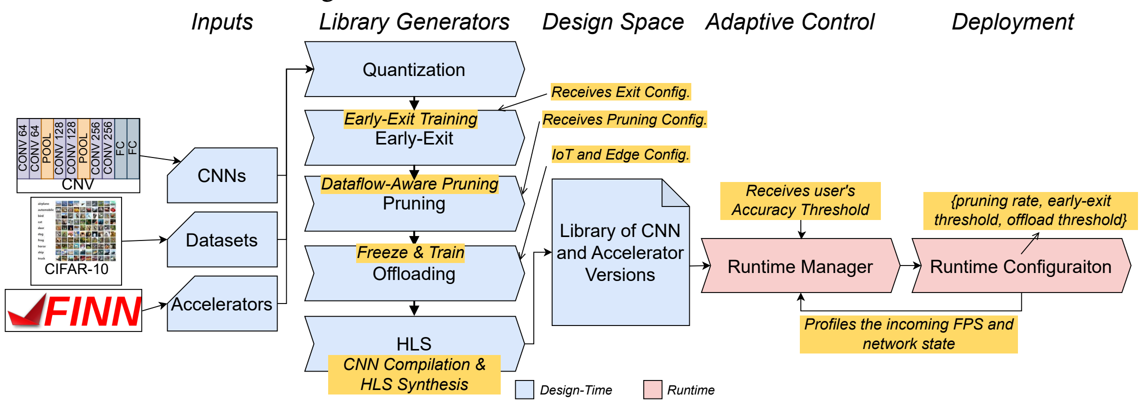


Figure 4.15 shows AOI's workflow. AOI represents this thesis final contribution and, thus, fills all the optimizations proposed in the base workflow from Figure 4.1 in the following aspects:

- The design-time Inputs step is kept as in the previous workfronts (receiving the CNV with CIFAR-10 dataset and FINN accelerator);
- Under the Library Generators step, the offloading optimization not only performs the split of the inference, but also performs a novel training mechanism that enables runtime adaptability on top of offloaded inferences (called Freeze & Train, presented next);
- The HLS step had to be extended as well to cover the a new hardware module required for treating offloaded feature maps. Additionally, different HLS configurations are used for the FPGAs on the IoT and edge server;
- After the steps from quantization to HLS, like in the earlier workfronts, a Library is produced to support the runtime adaptation - but *only the IoT device* adapts the running configuration (i.e., fixed edge server);
- The runtime configurations in AOI now include three parameters: the pruning rate and the early-exit confidence threshold (same as workfronts 1 in Section 4.2 and 2 in Section 4.3), and a new parameter developed here to control the rate of offloaded inferences based on the classification confidence (also presented next).

In this workfront, we consider that the IoT devices, subject to a more restricted environment, will be the ones adapting the running configuration. The CNN deployed at the edge server will be fixed, delivering inferences at a higher accuracy (given its larger FPGA and less restricted power and energy requirements). Nevertheless, the runtime steps taken here could easily be extended to the edge server. Below, we focus on the steps added by AOI to the general workflow.

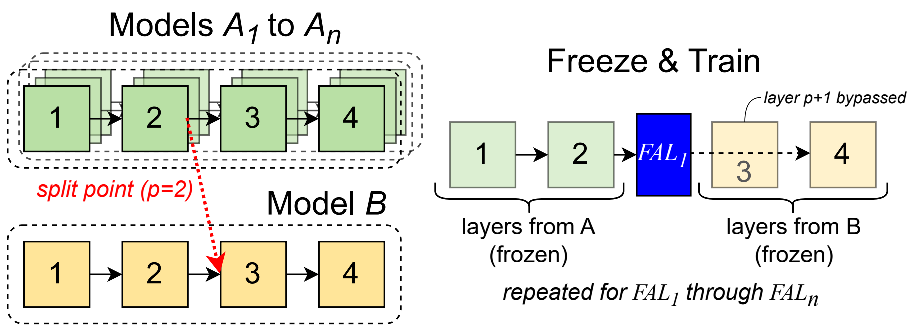
4.5.1.1 Library Generators

Quantization, Early-Exit, and Pruning. As stated before, AOI follows the procedure presented in Sections 4.3 and 4.2 for *quantizing*, adding *early-exits* and *pruning* the user's CNNs. The Offloading step, however, besides targeting IoT and edge FPGA devices (as in the previous workfront), also performs the Freeze & Train.

Freeze & Train. In the "traditional split approach" (recall Background Section 2.1.3.2), the inference is split from a single pre-trained CNN between the IoT device (running the initial layers) and an edge server (running the remaining layers). However, when switching layers to different pruning rates/quantization level, those are, in practice, running a different model (i.e., they were not trained together). That means that in the traditional split inference, feature maps (fmaps) from a pruned IoT device would not be understood by the receiving layers in the server side. To better illustrate the issue, we performed an initial experiment with the same CNV model used in the Sections 4.2 and 4.3. Offloading was set between feature maps from a 1-bit quantization early-exit CNV (to be deployed at the IoT) to a 2-bit quantization CNV (running on the edge server). The fmaps from the smaller model come from its first exit branch, after the second convolutional layer. They are then sent to the third layer of the larger, 2-bit quantization, CNV that would be deployed at the edge server. In this preliminary evaluation, no pruning was considered. This offloading resulted in a TOP-1 accuracy on the CIFAR-10 dataset of 10% (meaning a random classification). Therefore, by simply reducing from 2 to 1-bit quantization in the initial layers the classifications delivered with offloading become worthless.

To solve this issue, AOI employs a new training method: **Freeze & Train** (Figure 4.16). The goal of this method is to train Feature Adapter Layers (FAL) that will be deployed within the CNN on the edge server and are specialized to particular quantization, exit locations, and pruning rates of the IoT devices. Recall that after the pruning step in the workflow (Fig. 4.15), multiple versions of pruned early-exit CNNs are available to be switched at the IoT side - and will also be used to train different FALs. The method is inspired by the "classifier-only" methods for early-exits (LASKARIDIS; VENIERIS

Figure 4.16 – Freeze & Train for models A and B with adapter l_a.



et al., 2020). Here, however, instead of freezing a single set of layers in a single CNN, layers from different models are frozen.

Let us take as example a given CNN model A (with n multiple pruned versions) that will run on an IoT device and a CNN model B that will run on the edge server. Fig. 4.16 shows a possible split point ($p = 2$, i.e., fmap output by the second layer is offloaded) between A and B (the split point is valid for all n pruned versions of A). The Freeze & Train consists of placing one FAL (FAL_1) at the given split point $p + 1$ as depicted in the right size of Fig. 4.16. These adapters will be trained for each pruned version of A. In the forward pass, only the layers involved are used. In Fig. 4.16, that means inputs are fed to layers one and two of the CNN version A_1. Then, the inference continues into the adapter FAL_1 (in place of model B's layer 3) and later to the remaining layers in B. For the backward pass, however, only the adapter weights will be updated (leaving everything else in models A and B as is, or *frozen*).

The process is repeated for the remaining versions of A (i.e., each at a different pruning rate, models A_2 to A_n), each creating one adapter (FAL_2 to FAL_n, not in the figure). Later at runtime, these adapters will be available on the server side to process the incoming feature maps without the need to update the original CNN layers.

HLS. From an FPGA point of view, some additions were required in the HLS step from Figure 4.15. First, let us note that to configure the FAL adapter, we use the parameterization (i.e., number/size of CONV filters) of the layer at the split point p in B (e.g., B's layer 2 in Figure 4.16). In this way, both layer p and FAL layers produce feature maps of the same size in B (i.e., number of channels, quantization, etc.), facilitating the hardware implementation. Nevertheless, the feature map coming from A (FAL input) may still have a different depth due to pruned channels. For this reason, we need a dedicated hardware module.

Considering the pruning optimization used in this thesis, the feature maps coming from IoT nodes can vary only in the number of channels. For that, we leveraged the

Figure 4.17 – FAL schematics for the FPGA implementation.

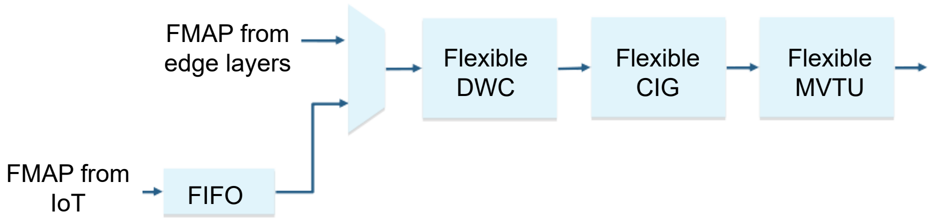
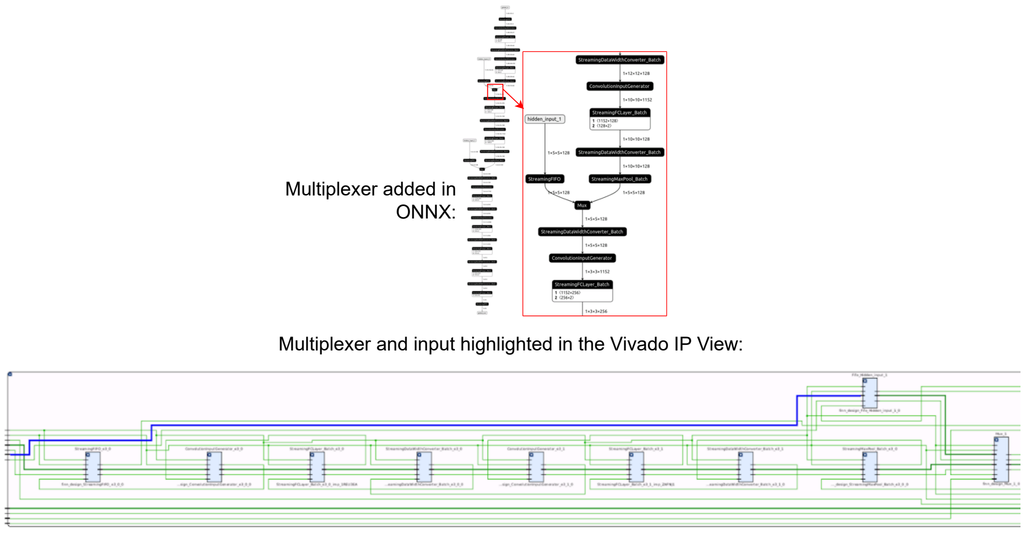
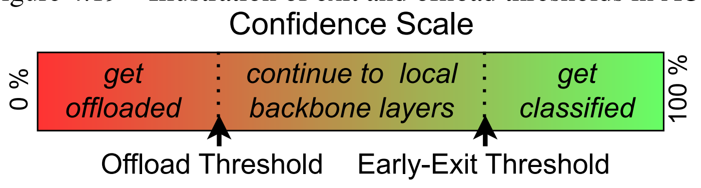


Figure 4.18 – Multiplexer for coupling the feature map adapter to remaining layers.



runtime adaptable HLS modules developed in the Pruning Workfront (AdaFlow in Section 4.2). Figure 4.17 shows the schematics used to implement FAL on FPGA. The *Flexible* modules are the ones leveraged from the Pruning Workfront. These modules allow us to change the pruning rate at runtime and, thus, to treat input fmaps of different number of channels. An additional modification was required: multiplexers selecting the entry point of the inference. It had be considered that the inference on the edge server can start with the first layer (original input) or start with the arrival of a feature map. To that end, we added to the FINN dataflow multiplexers that can be configured at runtime from the TOP interface. In Figure 4.17, the multiplexer selects between the input corresponding to "FMAP from edge server" (traditional operation) and the input corresponding to the offload with "FMAP from IoT." Figure 4.18 depicts this multiplexer implemented as a new FINN operator, and synthesized by the Vivado HLS. More implementation details on the FAL implementation are given in Appendices F (hardware) and G (software).

Figure 4.19 – Illustration of exit and offload thresholds in AOI.



4.5.1.2 Adaptive Control

In this workfront, we consider that the IoT devices will be the ones adapting the running configuration. Therefore the Runtime Manager in AOI runs on the IoT's board co-processor. As in the previous workfronts, the Runtime Manager works by continuously monitoring the current network state (i.e., latency and bandwidth values) and properties of the model versions in the Library produced at design-time. At every change in the sampled network state, the Runtime Manager searches for a new set of pruning, early-exit, and offload parameters (i.e., a new Runtime Configuration). As seen in the past workfronts, these parameters work as knobs for adapting the inference processing: the ***Pruning Rate*** controls the size of the CNN layers; the ***Early-Exit Threshold*** parameter controls how much of early classifications get accepted; and, the ***Offload Threshold*** is a new confidence-based threshold proposed here to decide whether to offload an inference on an input basis. Below, we give more details on the three runtime parameters[4].

Changing the Pruning Rate is the only parameter that requires FPGA reconfiguration, given that the FINN dataflow accelerators used in this evaluation are hardwired for a particular CNN model (traditional FINN accelerators). The other two act on the firmware running along the FPGA on the IoT device. The Early-Exit Threshold is the usual confidence threshold to decide whether an early classification should be accepted (see Background in Section 2.1.3.1). The Offload Threshold is a parameter proposed in AOI that uses the exit's classification confidence to decide whether or not the current sample should be offloaded to the edge server. The rationale behind this is that if, after a given exit, the confidence is still low (i.e., below the offload threshold), chances are the CNN at the IoT will not classify the sample correctly. In such cases, offloading that feature map to a more capable CNN running on the edge server presents a better chance of classifying the sample correctly. On a more practical note, having the offload controlled by a parameter allows AOI to set how much of the samples are offloaded and how much get classified locally. Combining the two thresholds creates a confidence scale (see Figure 4.19). The

[4]In this workfront, we will refer to the confidence threshold in early-exit CNNs as the "Early-Exit Threshold" to avoid confusion with the new confidence-based threshold that controls the offload.

classification confidence (highest softmax value) is taken at every early exit. AOI, then, measures it, first, against the exit threshold. If the classification is not finished, the confidence is compared to the offload threshold, sending only the most complex samples to the server. Intermediate confidence values (greater than the offload and lesser than the early-exit thresholds) continue into the local backbone layers remaining at the IoT device.

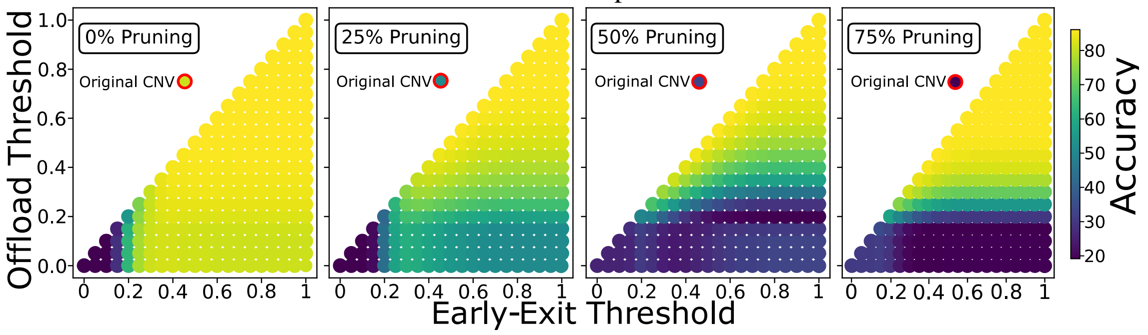
To find the pruning rate and exit/offload thresholds, the Runtime Manager searches the Library and takes the current networking latency and bandwidth values. The search works as the Runtime Mangers presented in the first two workfronts (Algorithm 2) with two small adaptations. First, in AOI, the Runtime Manager monitors the current networking delay and bandwidth to add it to the accelerator delay and throughput, respectively, before the search takes place. Second, in AOI, we added latency as a parameter to the optimization priority list P in Algorithm 2: $P = \{Latency, Throughput, Accuracy\}$. We note that, as in the previous workfronts, this search does not need to interrupt the inference processing and, therefore, does not cause overheads.

4.5.2 Results

4.5.2.1 Evaluation Scenario

Our use-case application is an IoT smart video application with an IoT device connected over cellular network to an edge server. We use a collection of 34 traces (measurement campaigns, around 10min each) from a dataset with real-world latency and bandwidth values (KOUSIAS et al., 2023). Traces were recorded on 4/5G clients moving around the city of Rome. Results are averaged over the 34 traces. Evaluations last the whole traces, with the IoT device producing 30 Inferences per Second (IPS) for the first half of the trace and 60 IPS for the second half to emulate higher workloads. Edge server targets a ZCU104 board (XCZU7EV) running the CNV CNN from FINN at 4-bit quantization (**CNVW4A4**), this was the largest CNV fitting the FPGA. The IoT device holds a Pynq-Z1 (XC7Z020) board. AOI generated for the IoT 4 pruned early-exit models (at 0, 25, 50, and 75% pruning rates) at 1-bit quantization (**CNVW1A1**). For 0% (not pruned) and 25% pruning, only one early exit fits the XC7Z020 FPGA. For 50 and 75% pruning, two early-exits are synthesized. Offload is placed at the first exit location. Confidence/offload thresholds vary from 0 to 100% at 5% steps. Accuracy is reported on Brevitas TOP-1 test accuracy. Training early-exit followed (TEERAPITTAYANON;

Figure 4.20 – Accuracy (color scale) w.r.t exit (x-axis) and offload (y-axis) thresholds for AOI inference with IoT layers at 0, 25, 50, and 75% pruning rates. Accuracy of pruned CNVW1A1 shown for comparison.



MCDANEL; KUNG, 2016), with 1.0 weight on the first exit and 0.3 to the remaining. Pruned models retrain for 40 epochs (LI et al., 2017) and FAL for 100 epochs.

Two baselines are used: **FINN IoT** (original CNV at 1-bit quantization running locally on the IoT FPGA) that represents an IoT-only SoTA solution; and **FINN Edge** that fully offloads all inferences (i.e. input images are sent) to the server running an original CNV at 4-bit quantization, representing a SoTA edge processing.

4.5.2.2 Evaluation

Table 4.6 – FPGA resource usage (*indicates the largest accelerator targeting the IoT board synthesized by AOI).

	FPGA	LUTs	FFs	BRAMs	Power (W)
FINN IoT	Pynq-Z1	31.22%	16.32%	46.43%	0.400
FINN Edge	ZCU104	27.20%	15.10%	68.75%	1.573
AOI	Pynq-Z1*	39.60%	21.4%	52.50%	0.409
	ZCU104	89.00%	22.90%	80.80%	2.797

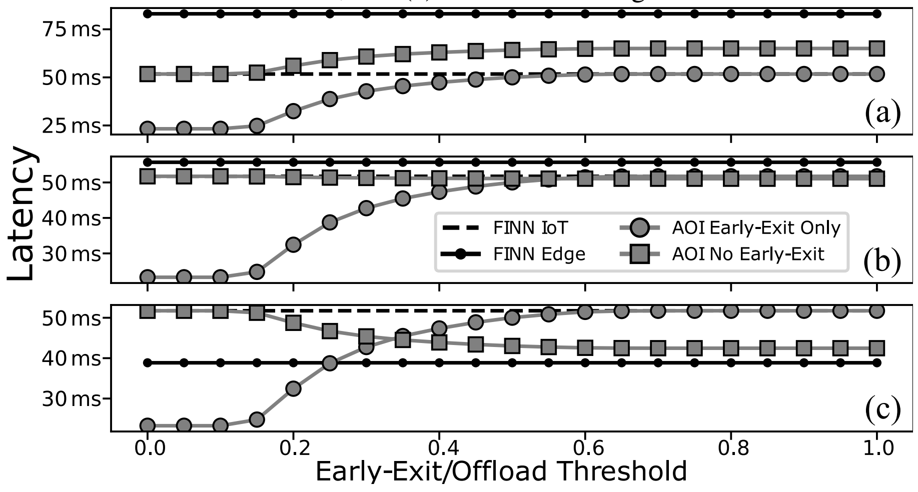
Resources. Table 4.6 shows the FPGA resource usage for the baselines and AOI designs. Note the AOI resource overhead (mainly LUTs and BRAMs) at both IoT (Pynq-Z1) and edge (ZCU104). AOI overhead at the IoT board (8.4 and 6.1% more LUTs and BRAMs than FINN IoT) is due to the early-exits circuitry. On the edge side, 57.8 and 34.37% more LUTs and BRAMs (w.r.t FINN Edge) are used but, in this case, they are due to the AOI's FALs (i.e., multiplexer and flexible HLS modules, recall Figure 4.17).

Accuracy. Figure 4.20 presents the AOI design space on accuracy for the evaluated dataset and CNN model. The figure gives a heatmap on accuracy (color scaled according to the color bar on the right, yellow the highest accuracy) w.r.t early-exit threshold (x-axis) and offload threshold (y-axis). Each combination of exit and offload thresholds returns a particular accuracy. You may also note that only points when the early-exit threshold is greater than or equal to the offload threshold are shown (recall the Confidence Scale in Figure 4.19). The AOI library contains four pruned IoT models from 0% (not-pruned version) to 75% pruning rate, and each has a plot in Fig. 4.20. The relatively high accuracy achieved by AOI (concentration of yellow points, meaning values above 80% accuracy), even for the heavily pruned versions, is noteworthy given the accuracy drop when pruning the original CNVW1A1 (highlighted in Fig. 4.20). The CNVW1A1 returns accuracy of 84.18, 72.55, 49.15, and 19.55% when pruned at 0, 25, 50, and 75% rates, respectively (considerably below most of the AOI design points). The higher accuracy in AOI is due to the split inference between the CNVW1A1 and CNVW4A4 that runs on the edge server. You may note, for instance, a growing number of yellow points as the y-axis (offload threshold) increases. Recall from Subsection 4.5.1.2 that inference is offloaded when the confidence value is *below* the offload threshold. Therefore, by increasing the offload threshold, more samples get offloaded and classified at the edge model, which delivers a higher accuracy.

The early-exit threshold also impacts the accuracy. By increasing the exit threshold, more samples get classified at the last exit (behavior regularly found in any early-exit and in Section 4.3). For example, on the 0% pruning rate (leftmost plot), let us take the first row as an example (0% offload threshold). As we move on to the larger early-exit thresholds (from left to right on the x-axis), we see an increase in accuracy (more yellow points). This increase is not due to more offloaded samples since, for this row (y-axis at 0%), there is no offloading happening - no confidence can get below the offload threshold of 0%. The increase is due to more samples getting classified at the last exit in the IoT CNN. Now, if we do the same for 75% pruning rate (right-most plot), there is no such accuracy increase on the first row (0% offload threshold). This happens because the IoT backbone was heavily (75%) pruned and, therefore, is unable to deliver high-accuracy classifications. Nevertheless, having pruned models running on the IoT introduces interesting design points on the latency-accuracy trade-off.

Latency. Figure 4.21 gives the latency on a single inference for the baselines running the CNVW1A1 on FINN IoT (dashed curve) and running the CNVW4A4 on the FINN Edge (black solid curve), and, for the sake of simplicity, two representative platform configurations of AOI with two extreme cases: the *Early-Exit Only* (i.e., AOI with offload threshold set to zero), causing all inferences to happen at the IoT device (e.g., when no IoT-edge communication is available); and the *No Early-Exit* (i.e., AOI with early-exit

Figure 4.21 – Latency for FINN IoT, FINN Edge, and two AOI configurations under (a) best, (b) mean, and (c) worst networking.



threshold set to 100%) causing samples to get either offloaded or classified at the IoT last exit (no early classifications since no confidence will be above 100%). The x-axis represents variations on either the early-exit threshold (for AOI's *Early-Exit Only*) or the offload threshold (for AOI's *No Early-Exit*). Each subplot in Figure 4.21 gives a corner case on the distribution of the recorded 4/5G networking quality: (a) is produced with the 1% best values from the recorded latency, (b) uses the recorded mean latency, and (c) uses the 1% tail latency. On comparing FINN IoT (always local) against FINN Edge (full offload), we note that the configuration giving the fastest inference depends heavily on the communication costs. As FINN Edge offloads the full input image, it ends up with latency around 50% higher than the FINN IoT when the network is at its worst (plot a). In the best networking case (plot c), however, FINN Edge achieves latency around 25% better than FINN IoT. *Therefore, relying on a fixed solution is unfeasible considering the highly unpredictable IoT-edge environment.*

On plots (a) and (b), for worst and average networking quality, AOI's *Early-Exit Only* gives better or equal latency than AOI's *No Early-Exit*. This is primarily because relying on a local early-exit gives fast classifications free from any networking delays. Similarly to what happens on accuracy, as the early-exit threshold increases, more inferences continue into the IoT final layers, causing an increase in latency. Only when networking is at its best condition (plot c), we have a turning point around thresholds of 35% where the *No Early-Exit* becomes faster. As in this corner case, communication costs decrease significantly, and offloading feature maps becomes worthwhile for delivering a lower latency. In summary, it becomes clear that static and optimized inference processing can deliver low latency *or* high accuracy but will hardly cover all behaviors required at

Table 4.7 – Results averaged over the 34 evaluated traces.

	Accuracy	Inference Loss	Latency	Norm. Frames/Watt
FINN IoT	84.18%	32.01%	51.74ms	1.00
FINN Edge	**90.37%**	0.39%	55.40ms	0.37
AOI	84.70%	**0.07%**	**34.8ms**	**1.46**

runtime. Therefore, these optimizations must be jointly exploited in an adaptive manner for the IoT-edge continuum.

Results under an application scenario. Table 4.7 summarizes the use evaluation scenario of an IoT smart video surveillance (presented in Subsection 4.5.2.1). The table gives the results averaged over 34 traces on TOP-1 accuracy, inference loss (the rate of lost inferences), latency, and power efficiency (i.e., frames per Watt) normalized to the FINN IoT. Compared to state-of-the-art FINN accelerator working locally (FINN IoT) or processing the offloaded input images on the edge server (FINN Edge), AOI delivers better levels of Inference Loss (losing only 0.07% of the workload inferences) and better latency (processing inferences in 34.8ms on average). Thanks to the runtime AOI adaptation, it can tune the parameters of the pruning rate and exit/offload thresholds according to the current networking conditions. As a result, AOI balances out the high throughput of the edge server (returning minimal inference loss) and the low latency delivered by the local IoT FPGA. This balancing also causes AOI to deliver the best power efficiency. Even having FPGAs dissipating power at both IoT and server ends, AOI is $1.46\times$ and $3.9\times$ more efficient than the FINN IoT and Edge baselines, respectively.

The highest average accuracy, however, is delivered by the FINN Edge, which runs the whole inference on the largest model (CNVW4A4). Nevertheless, when compared to FINN IoT, which also runs an original CNN but at 1-bit quantization, AOI does not incur any accuracy loss (0.52% higher accuracy). This slight improvement in accuracy over FINN IoT happens at the same time that latency is significantly reduced. Considering all evaluation runs, AOI used 19 unique sets of parameters (pruning rate, exit, and offloaded thresholds). These values span from 0% to 50% pruning rates, exit thresholds from 35 to 100%, and offload thresholds from 10 to 100%. Changes in pruning rate required an average of only six FPGA reconfigurations (around 900ms) for each evaluation run (trace), resulting in virtually no impact on the minutes-long evaluations.

5 CONCLUSION

Before we discuss the thesis final remarks, we first present some aspects that we understand as current limitations of this work and ideas on how to extend this thesis' work.

5.1 Limitations

On the pruning technique. Filters are pruned in this thesis based on their floating-point representation ("Dataflow-Aware Pruning" in Section 4.2). While we have seen that this pruning technique has presented acceptable results on accuracy, we believe that relying on the weights' importance (measured from the filters $l1-$norm) before quantization may not be the optimal approach to evaluate the filter importance. As noted in (FARAONE et al., 2018), quantizing weights to 1 or 2 bits may create significant similarity between filters (i.e., values that were different before quantization end up with the same quantized value). Our pruning method ignores this characteristic of quantized CNNs. Including it in our pruning technique could further improve our results. This could be achieved by changing the importance measure by, for example, ranking the filters according to their *uniqueness* or *similarity*.

On the implementation of the Flexible HLS modules. The overhead crated to have the flexible HLS modules in Section 4.2 is noteworthy (roughly doubling the LUT usage). As discussed throughout the text, the flexible HLS modules are capable of changing the pruning rate at runtime and, in order to do that, have some extra control logic in the HLS modules in charge of processing convolutional layers. Here, we point out that further investigations could shed some light on how to decrease such overheads. For example, a more elegant solution could avoid our current `if`-controlled loops, easing the HLS unrolling/pipelining to reduce the cost of the control logic. More details on the FPGA implementation are offered in Appendix F.

On the FPGA implementation of early-exit. Having the CNN implemented as a dataflow on the FPGA allowed us to unfold the multiple branches on the FPGA. Concurrently feeding both backbone and early exits increases parallelism, amortizing the early exit overhead. However, when taken, these exits are *preempted* early from the backbone, creating 'bubbles' in the dataflow pipeline since its feature maps continue onto the backbone layers, not creating useful outputs. Kouris et al. noted this behavior in a software implementation (running on GPU) (KOURIS et al., 2022). In their case, the feature maps do not continue following the dataflow, but instead, their processing is interrupted from the GPU, which causes the emptying of the GPU memory hierarchy, possibly affecting performance. They solve that issue by adapting the batch size and holding some inputs to feed at the early exit points to keep a filled batch at all times. Addressing this limitation in our implementation could further improve the performance of early-exit CNNs on FPGA.

On the offloading technique. The offloading decision proposed in Section 4.5 has the advantage of being dynamic and working in a per-sample basis. However, relying on a confidence metric implies that an early-exit must be added to the CNN. On top of that, it also means that the set of possible split locations is restricted to the layers where an exit is placed. Therefore, it may 'hide' an optimal split location placed after a non-branching layer in the CNN. A study considering every layer of the CNN as a possible split position is offered in Appendix E. As seen in this evaluation, the position of optimal offloading can be always found by navigating the design space of all possible offloading configurations, which, in turn, could be used to place the exits. Regardless, we consider that it brings an additional difficulty to the designer and is, therefore, a limitation of the proposed offloading mechanism.

On the evaluation methodology. To speed up the development of this thesis, we have opted to evaluate our work with RTL simulations. Despite giving accurate performance measures of the accelerator designs, it fails to give a proper assessment on memory access. In our works, we have paid attention to this limitation in two ways. First, with single-engine accelerators (works in the Appendices A, B, and C), we use SDx tool from Xilinx. This tool fully profiles the design's memory interface, with latency and throughput values. We have included this data in our results. Second, with the main workfronts that use the FINN design flow and, therefore, are synthesized with Vivado, we have no access to such reports. For those works, we have considered the board nominal memory bandwidth and the reports from RTL simulations to guarantee that the rate of memory reads required at inference do not exceed the board's nominal bandwidth, ensuring that no memory stall is inserted in the dataflow.

On the evaluation use cases. In this thesis, we have focused on CNNs in all evaluation cases. While they still are the SoTA for many applications, especially in IoT/embedded, the rising popularity of large language models (LLM) applications, not enabled by CNNs anymore, is evident. Tools like ChatGPT (LIU et al., 2023) bring the question of how and when those technologies will be deployed closer to the user - instead of residing only in the cloud, as done today. In this regard, we first need to mention that we restricted our use cases to the ML models supported by the compilation tools (e.g., FINN). In fact, a generic method for compiling ML models to efficient FPGA accelerators is an open research problem. Secondly, we want to note that, despite the significant difference in problem size (PaLM has 540B parameters (CHOWDHERY et al., 2022)), both CNNs and LLMs heavily rely on matrix multiplications, allowing some of the takeaways from this thesis to be extrapolated to more modern models.

5.2 Future Works

On the versioning of CNNs and accelerators. Design methods like Automated Machine Learning (AutoML) (HE; ZHAO; CHU, 2021) and Neural Architecture Search (NAS) (TERMRITTHIKUN et al., 2021; ZHAO et al., 2021) have shown great success in navigating the design space of deep learning inference. Despite being traditionally used as a static method (i.e., finding a particular configuration before deployment or even before training in some cases), such methods could be used in this work to generate the CNN/accelerator versions in the Library supporting the runtime adaptation steps. For instance, instead of having the optimization parameters ranging at fixed intervals to generate the Library (as done in the workfronts from Chapter 4), one could use AutoML/NAS to generate a handful of versions each representing a particular accuracy-performance-energy trade-off.

On the optimization goals. Throughout this work, we have used some combination of performance (latency or throughput), energy, and power as the optimization goal. This combination represents a multi-objective search that was solved as a lexicographic search (for the workfronts with runtime adaptability) and by sorting a weighted profit equation (in the workfront without the runtime phase). In all cases, accuracy was used as a constraint (i.e., by setting the Accuracy Threshold). A possible future work resides in extending the set of objective goals and constraints to other metrics. For instance, it is possible to use latency as a constraint (e.g., a Latency Threshold). Such a new constraint could return a not-yet-explored part of the design space and, in turn, produce design points with a different performance-cost profile.

On new pruning methods. The pruning method developed for this work considers the same pruning rate across all convolutional layers of the CNN. However, it is known (LI et al., 2017) that different convolutional layers present different resiliency to pruning and, thus, different accuracy costs. One possible future work consists of considering this behavior to define pruning rates on a per-layer basis. Such pruning could also consider not only the accuracy aspect but also eventual bottleneck layers to apply a more severe pruning rate - only when necessary.

5.3 Final Remarks

This thesis proposed approaching FPGA-based CNN inferences from various optimization angles. We focused on the IoT-edge paradigm, which offers the exciting challenge of high-performance demands with restricted resources. Motivated by the flourishing use of FPGAs for CNN acceleration, we saw, at the beginning of this research, an opportunity to match the (re-)configurability enabled by FPGAs with the CNN optimizations being studied by the deep learning community. It is clear that there is a need for faster and cheaper inferences. However, there is still no obvious winner for carrying out these optimizations. In the multiple stages of this work, we have investigated many possible approaches, from approximate accelerators to modifications on the CNN topologies like pruning, early-exit and split inferences. In this context, this thesis offered us with a better understanding of how to trade-off impacts on quality (e.g., accuracy drops) for better performance and efficiency levels. These observations supported the construction of an integrated framework that combines optimizations at multiple levels for delivering fully configurable and adaptable inference processing.

5.4 Publications

The following works have been published in the scope of this thesis:

1. *Synergistically Exploiting CNN Pruning and HLS Versioning for Adaptive Inference on Multi-FPGAs at the Edge*. Guilherme Korol, Michael Jordan, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at CASES 2021 (published in ACM TECS).
2. *ConfAx: Exploiting Approximate Computing for Configurable FPGA CNN Acceleration at the Edge*. Guilherme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at ISCAS 2022.
3. *AdaFlow: A Framework for Adaptive Dataflow CNN Acceleration on FPGAs*. Guil-

herme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at DATE 2022 (**Best Paper Nominee**).

4. *Pruning and Early-Exit Co-Optimization for CNN Acceleration on FPGAs*. Guilherme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, Jeronimo Castrillon, and Antonio Carlos Schneider Beck at DATE 2023.
5. *Design Space Exploration for CNN Offloading to FPGAs at the Edge*. Guilherme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, Jeronimo Castrillon, and Antonio Carlos Schneider Beck at ISVLSI 2023.
6. *Adaptive and Offloaded CNNs for IoT-Edge FPGAs*. Guilherme Korol and Antonio Carlos Schneider Beck at ISVLSI 2024 (under review).
7. *Exploiting Pruning and Quantization for FPGA-Based Split Inference*. Guilherme Korol and Antonio Carlos Schneider Beck at IEEE Transactions on VLSI (under review).
8. *Adaptive Inference for FPGA-based 5G Automatic Modulation Classification*. Daniel Rubiano, Guilherme Korol, and Antonio Carlos Schneider Beck at DASIP 2023 (published in Lecture Notes in Computer Science).

Besides the above publications, the following works were also published during the author's time as a Ph.D. student from collaborations with other students in the group.

9. *Dynamic Offloading for Improved Performance and Energy Efficiency in Heterogeneous IoT-Edge-Cloud Continuum*. Julio Costella Vicenzi, Guilherme Korol, Michael G. Jordan, Wagner Ourique de Morais, Hazem Ali, Edison Pignaton de Freitas,Mateus Beck Rutzig, Antonio Carlos Schneider Beck at ISVLSI 2023.
10. *Harnessing the Effects of Process Variability to Mitigate Aging in Cloud Servers*. Arthur F. Lorenzon, Guilherme Korol, Marcelo Brandalero, Antonio Carlos Schneider Beck at ISVLSI 2023.
11. *Resource Provisioning for CPU-FPGA Environments with Adaptive HLS-Versioning and DVFS*. Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at ISVLSI 2023.
12. *Adaptive Inference on Reconfigurable SmartNICs for Traffic Classification*. Julio Costella Vicenzi, Guilherme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at AINA 2023.
13. *On the benefits of Collaborative Thread Throttling and HLS-Versioning in CPU-FPGA Environments*. Tiago Knorst, Guilherme Korol, Michael Guilherme Jordan,

Julio Costella Vicenzi, Arthur Lorenzon, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at SBCCI 2022.

14. *ERIN: Energy-Aware Resource Provisioning Framework for CPU-FPGA Multi-tenant Environment*. Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at IEEE Design & Test 2022.
15. *An energy efficient multi-target binary translator for instruction and data level parallelism exploitation*. Tiago Knorst, Julio Vicenzi, Michael G Jordan, Jonathan H de Almeida, Guilherme Korol, Antonio Beck, and Mateus B Rutzig at Design Automation for Embedded Systems 2022.
16. *TRIPP: Transparent Resource Provisioning for Multi-Tenant CPU-GPU based Cloud Environments*. Julio Costella Vicenzi, Tiago Knorst, Michael G Jordan, Guilherme Korol, Antonio Carlos Schneider Beck, and Mateus Beck Rutzig at SBESC 2021 (**Best Paper Nominee**).
17. *FAIR: Fully-Adaptive Framework for Improving Resource Provisioning in Collaborative CPU-FPGA Cloud Environments*. Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at SBAC-PAD 2021.
18. *Muteco: A framework for collaborative allocation in cpu-fpga multi-tenant environments*. Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at SBCCI 2021 (**Best Paper Nominee**).
19. *Dynamic concurrency throttling on numa systems and data migration impacts*. Janaina Schwarzrock, Michael Guilherme Jordan, Guilherme Korol, Charles C de Oliveira, Arthur F Lorenzon, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at Design Automation for Embedded Systems 2021.
20. *Resource-Aware Collaborative Allocation for CPU-FPGA Cloud Environments*. Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at IEEE Trans. on Circuits and Systems II: Express Briefs 2021.
21. *Exploiting HLS-Generated Multi-Version Kernels to Improve CPU-FPGA Cloud Systems*. Bernardo Neuhaus Lignati, Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck at ASP-DAC 2021.
22. *A Management Technique for Concurrent Access to a Reconfigurable Accelerator*. Raul Silva, Guilherme Korol, Michael Guilherme Jordan, Marcelo Brandalero, Michael Hübner, Monica Pereira, Mateus Beck Rutzig, and Antonio Carlos Schnei-

der Beck at SBCCI 2020.

23. *Unlocking the Full Potential of Heterogeneous Accelerators by Using a Hybrid Multi-Target Binary Translator*. Tiago Knorst, Julio Vicenzi, Michael Guilherme Jordan, Jonathan Almeida, Guilherme Korol, Antonio Carlos Schneider Beck, Mateus Beck Rutzig at SBCCI 2020.
24. *Firefly: An Open-source Rocket-based Intermittent Framework*. Hiago Rocha, Guilherme Korol, Michael Jordan, Arthur Krause, Ronaldo Silveira, Caio Vieira, Philippe Navaux, Gabriel L Nazar, Luigi Carro, and Antonio Carlos Schneider Beck at SBCCI 2020.

REFERENCES

AARRESTAD, T. et al. Fast convolutional neural networks on fpgas with hls4ml. **CoRR**, abs/2101.05108, 2021.

ADEGBIJA, T. et al. Microprocessor optimizations for the internet of things: A survey. **IEEE Trans. on CAD of IC and Systems**, v. 37, n. 1, p. 7–20, 2018.

Aldec. **Deep Learning Using Zynq US+ FPGA**. 2023. <https://www.aldec.com/en/solutions/embedded/deep-learning-using-fpga>.

Amazon. **Amazon EC2 F1 Instances**. 2023. <https://aws.amazon.com/ec2/instance-types/f1/>.

AMAZON, E. Amazon ec2 f1 instances. **Disponível em: https://aws.amazon.com/ec2/instance-types/f1/ (May 2023)**, 2023.

AMIRI, S. et al. Multi-precision convolutional neural networks on heterogeneous hardware. In: IEEE. **2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2018. p. 419–424.

AYHAN, T.; ALTUN, M. Circuit aware approximate system design with case studies in image processing and neural networks. **IEEE Access**, v. 7, p. 4726–4734, 2019.

BAILEY, D. G. **Design for embedded image processing on FPGAs**. [S.l.]: Wiley-IEEE Press, 2011.

BASKIN, C. et al. Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform. In: **IPDPS**. [S.l.: s.n.], 2018.

BENGIO, Y.; LÉONARD, N.; COURVILLE, A. C. Estimating or propagating gradients through stochastic neurons for conditional computation. **CoRR**, abs/1308.3432, 2013.

BIGGS, B.; BOUGANIS, C.-S.; CONSTANTINIDES, G. A. Atheena: A toolflow for hardware early-exit network automation. **arXiv preprint arXiv:2304.08400**, 2023.

BINKERT, N. L. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, v. 39, n. 2, p. 1–7, 2011.

BLOTT, M. et al. Finn-*R*: An end-to-end deep-learning framework for fast exploration of quantized neural networks. **ACM TRETS**, v. 11, n. 3, p. 16:1–16:23, 2018.

BROWN, R. J. F. S. D. et al. **Field-Programmable Gate Arrays**. [S.l.]: Springer, 1992.

CAPOTONDI, A. et al. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, v. 67, n. 5, p. 871–875, 2020.

CARTAS, A. et al. A reality check on inference at mobile networks edge. In: **EdgeSys**. [S.l.]: ACM, 2019. p. 54–59.

CASTRO-GODÍNEZ, J. et al. Approximate acceleration for cnn-based applications on iot edge devices. In: **LASCAS**. [S.l.]: IEEE, 2020. p. 1–4.

CHAKRADHAR, S. T.; RAGHUNATHAN, A. Best-effort computing: re-thinking parallel software and hardware. In: **DAC**. [S.l.]: ACM, 2010. p. 865–870.

CHEN, F. et al. Enabling fpgas in the cloud. In: **Proceedings of the 11th ACM Conference on Computing Frontiers**. [S.l.: s.n.], 2014. p. 1–10.

CHEN, J.; RAN, X. Deep learning with edge computing: A review. **Proc. IEEE**, v. 107, n. 8, p. 1655–1674, 2019.

CHEN, X. et al. Energy-efficient offloading for dnn-based smart iot systems in cloud-edge environments. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 33, n. 3, p. 683–697, 2021.

CHETLUR, S. et al. cudnn: Efficient primitives for deep learning. **arXiv preprint arXiv:1410.0759**, 2014.

CHOLLET, F. Xception: Deep learning with depthwise separable convolutions. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2017. p. 1251–1258.

CHOUDHARY, T. et al. A comprehensive survey on model compression and acceleration. **Artif. Intell. Rev.**, v. 53, n. 7, p. 5113–5155, 2020.

CHOWDHERY, A. et al. Palm: Scaling language modeling with pathways. **arXiv preprint arXiv:2204.02311**, 2022.

Coelho Jr, C. N. et al. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. **Nature Machine Intelligence**, Nature Publishing Group UK London, v. 3, n. 8, p. 675–686, 2021.

CONSTANTINIDES, G. A. Fpgas in the cloud. In: **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2017. p. 167–167.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J. Binaryconnect: Training deep neural networks with binary weights during propagations. In: **NIPS**. [S.l.: s.n.], 2015. p. 3123–3131.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Training deep neural networks with low precision multiplications. **arXiv preprint arXiv:1412.7024**, 2014.

COURBARIAUX, M. et al. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. **arXiv preprint**, 2016.

CRANKSHAW, D. et al. Clipper: A low-latency online prediction serving system. In: **USENIX-NSDI**. [S.l.: s.n.], 2017. p. 613–627.

DENIL, M. et al. Predicting parameters in deep learning. In: **NIPS**. [S.l.: s.n.], 2013. p. 2148–2156.

DU, X. et al. Fused DNN: A deep neural network fusion approach to fast and robust pedestrian detection. In: **WACV**. [S.l.]: IEEE Computer Society, 2017. p. 953–961.

European Commission for Comm. Networks, Content and Technology. **Study on the economic potential of far edge computing in the future smart Internet of Things – Final study report**. [S.l.]: Publications Office of the European Union, 2023.

FANG, B. et al. FlexDNN: Input-Adaptive On-Device Deep Learning for Efficient Mobile Vision. In: **SEC**. [S.l.]: IEEE, 2020. p. 84–95.

FANG, B.; ZENG, X.; ZHANG, M. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In: **MobiCom**. [S.l.: s.n.], 2018. p. 115–127.

FARABET, C. et al. Cnp: An fpga-based processor for convolutional networks. In: IEEE. **2009 International Conference on Field Programmable Logic and Applications**. [S.l.], 2009. p. 32–37.

FARAONE, J. et al. Customizing low-precision deep neural networks for fpgas. In: **FPL**. [S.l.: s.n.], 2018.

FARHADI, M.; GHASEMI, M.; YANG, Y. A novel design of adaptive and hierarchical convolutional neural networks using partial reconfiguration on FPGA. In: **HPEC**. [S.l.]: IEEE, 2019. p. 1–7.

FRANKLE, J.; CARBIN, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. **arXiv preprint arXiv:1803.03635**, 2018.

FU, B. et al. A survey of cross-layer designs in wireless networks. **IEEE Commun. Surv. Tutorials**, v. 16, n. 1, p. 110–126, 2014.

GHOLAMI, A. et al. A survey of quantization methods for efficient neural network inference. **CoRR**, abs/2103.13630, 2021.

Global Market Insights. **Data Center Accelerator Market**. 2023. <https://www.gminsights.com/industry-analysis/data-center-accelerator-market>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016.

GRIFFIN, G.; HOLUB, A.; PERONA, P. Caltech-256 object category dataset. **Technical Report**, California Institute of Technology, 2007.

GUO, C. et al. On calibration of modern neural networks. In: PMLR. **International conference on machine learning**. [S.l.], 2017. p. 1321–1330.

GUO, K. et al. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. **IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.**, v. 37, n. 1, p. 35–47, 2018.

GUO, Y.; YAO, A.; CHEN, Y. Dynamic network surgery for efficient dnns. In: **NIPS**. [S.l.: s.n.], 2016. p. 1379–1387.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv**, 2016.

HAN, S. et al. Learning both weights and connections for efficient neural networks. In: **NIPS**. [S.l.: s.n.], 2015. p. 1135–1143.

HAN, S. et al. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In: **Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services**. [S.l.: s.n.], 2016. p. 123–136.

HAO, C. et al. FPGA/DNN co-design: An efficient design methodology for iot intelligence on the edge. In: **DAC**. [S.l.: s.n.], 2019.

HAUSWALD, J. et al. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In: **ISCA**. [S.l.]: ACM, 2015. p. 27–40.

HAWKS, B. et al. Ps and qs: Quantization-aware pruning for efficient low latency neural network inference. **Frontiers in Artificial Intelligence**, Frontiers Media SA, v. 4, p. 676564, 2021.

HE, K.; ZHANG, X. et al. Deep residual learning for image recognition. In: **IEEE Conference on Computer Vision and Pattern Recognition, CVPR**. [S.l.: s.n.], 2016. p. 770–778.

HE, X.; ZHAO, K.; CHU, X. Automl: A survey of the state-of-the-art. **Knowl. Based Syst.**, v. 212, p. 106622, 2021.

HE, X.; ZHOU, Z.; THIELE, L. Multi-task zipping via layer-wise neuron sharing. In: **NeurIPS**. [S.l.: s.n.], 2018. p. 6019–6029.

HINTON, G. E.; VINYALS, O.; DEAN, J. Distilling the knowledge in a neural network. **CoRR**, abs/1503.02531, 2015.

HOOFT, J. van der et al. HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks. **IEEE Communications Letters**, IEEE, v. 20, n. 11, p. 2177–2180, 2016.

HOOKER, S. The hardware lottery. **Communications of the ACM**, ACM New York, NY, USA, v. 64, n. 12, p. 58–65, 2021.

HOUBEN, S. et al. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In: **IJCNN**. [S.l.]: IEEE, 2013. p. 1–8.

HUANG, G. et al. Multi-scale dense networks for resource efficient image classification. **arXiv preprint arXiv:1703.09844**, 2017.

HUANG, G. et al. Densely connected convolutional networks. In: **CVPR**. [S.l.]: IEEE Computer Society, 2017. p. 2261–2269.

HUBARA, I. et al. Binarized neural networks. **Advances in neural information processing systems**, v. 29, 2016.

HUYNH-THU, Q.; GHANBARI, M. Temporal aspect of perceived quality in mobile video broadcasting. **IEEE Trans. Broadcast.**, v. 54, n. 3, p. 641–651, 2008.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: PMLR. **International conference on machine learning**. [S.l.], 2015. p. 448–456.

JACOB, B. et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: **CVPR**. [S.l.]: Computer Vision Foundation / IEEE Computer Society, 2018. p. 2704–2713.

JACOB, B. et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2018. p. 2704–2713.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. **arXiv**, 2014.

JIANG, S. et al. Accelerating mobile applications at the network edge with software-programmable fpgas. In: **INFOCOM**. [S.l.]: IEEE, 2018. p. 55–62.

JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In: **Proceedings of the 44th annual international symposium on computer architecture**. [S.l.: s.n.], 2017. p. 1–12.

KANG, W.; KIM, D.; PARK, J. DMS: dynamic model scaling for quality-aware deep learning inference in mobile and embedded devices. **IEEE Access**, v. 7, p. 168048–168059, 2019.

KANG, Y. et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In: **ASPLOS**. [S.l.: s.n.], 2017. p. 615–629.

KAYA, Y.; HONG, S.; DUMITRAS, T. Shallow-deep networks: Understanding and mitigating network overthinking. In: **ICML**. [S.l.]: PMLR, 2019. (Proceedings of Machine Learning Research, v. 97), p. 3301–3310.

KAYA, Y.; HONG, S.; DUMITRAS, T. Shallow-deep networks: Understanding and mitigating network overthinking. In: PMLR. **International conference on machine learning**. [S.l.], 2019. p. 3301–3310.

KHANH, P. N. et al. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In: **DATE**. [S.l.]: ACM, 2015. p. 157–162.

KONG, M.; NIKOV, K.; NUNEZ-YANEZ, J. L. Evaluation of early-exit strategies in low-cost fpga-based binarized neural networks. In: IEEE. **2022 25th Euromicro Conference on Digital System Design (DSD)**. [S.l.], 2022. p. 01–08.

KONG, M.; NUNEZ-YANEZ, J. L. Entropy-based early-exit in a fpga-based low-precision neural network. In: SPRINGER. **Applied Reconfigurable Computing. Architectures, Tools, and Applications: 18th International Symposium, ARC 2022, Virtual Event, September 19–20, 2022, Proceedings**. [S.l.], 2022. p. 72–86.

KOROL, G. et al. Design space exploration for CNN offloading to fpgas at the edge. In: **ISVLSI**. [S.l.]: IEEE, 2023. p. 1–6.

KOROL, G. et al. Pruning and early-exit co-optimization for CNN acceleration on fpgas. In: **DATE**. [S.l.]: IEEE, 2023. p. 1–6.

KOROL, G. et al. Adaflow: A framework for adaptive dataflow CNN acceleration on fpgas. In: **DATE**. [S.l.]: IEEE, 2022. p. 244–249.

KOURIS, A. et al. Fluid batching: Exit-aware preemptive serving of early-exit neural networks on edge npus. **arXiv preprint**, 2022.

KOUSIAS, K. et al. A large-scale dataset of 4g, nb-iot, and 5g non-standalone network measurements. **IEEE Communications Magazine**, IEEE, 2023.

KRISHNAMOORTHI, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. **CoRR**, abs/1806.08342, 2018.

KRIZHEVSKY, A.; HINTON, G. Learning multiple layers of features from tiny images. **Technical Report**, Citeseer, 2009.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **NIPS**. [S.l.: s.n.], 2012. p. 1106–1114.

LAI, L.; SUDA, N.; CHANDRA, V. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. **arXiv preprint arXiv:1801.06601**, 2018.

LASKARIDIS, S.; KOURIS, A.; LANE, N. D. Adaptive inference through early-exit networks: Design, challenges and directions. In: **EMDL@MobiSys**. [S.l.]: ACM, 2021. p. 1–6.

LASKARIDIS, S. et al. SPINN: synergistic progressive inference of neural networks over device and cloud. In: **MobiCom**. [S.l.]: ACM, 2020. p. 1–15.

LASKARIDIS, S.; VENIERIS, S. I. et al. HAPI: hardware-aware progressive inference. In: **ICCAD**. [S.l.]: IEEE, 2020. p. 91:1–91:9.

LEONTIADIS, I. et al. It's always personal: Using early exits for efficient on-device CNN personalisation. In: **HotMobile**. [S.l.]: ACM, 2021. p. 15–21.

LI, F. et al. Ternary weight networks. **arXiv preprint arXiv:1605.04711**, 2016.

LI, H. et al. Pruning filters for efficient convnets. In: **ICLR**. [S.l.]: OpenReview.net, 2017.

LIN, Z. et al. Neural networks with few multiplications. **arXiv preprint arXiv:1510.03009**, 2015.

LIU, Y. et al. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models. **arXiv preprint arXiv:2304.01852**, 2023.

LOU, W. et al. Dynamic-ofa: Runtime DNN architecture switching for performance scaling on heterogeneous embedded platforms. In: **CVPR Workshops**. [S.l.: s.n.], 2021.

MCCARTHY, J. What is artificial intelligence. Stanford University, 2007.

Microsoft. **Project Brainwave**. 2023. <https://www.microsoft.com/en-us/research/project/project-brainwave/>.

MICROSOFT. Project catapult. **Disponível em: https://www.microsoft.com/en-us/research/project/project-catapult/ (May 2023)**, 2023.

Mipsology. **FASTER AI INFERENCE COMPUTATION**. 2023. <https://mipsology.com/>.

MISHRA, R.; GUPTA, H. P.; DUTTA, T. A survey on deep neural network compression: Challenges, overview, and solutions. **CoRR**, abs/2010.03954, 2020.

MITCHELL, T. M. et al. **Machine learning**. [S.l.]: McGraw-hill New York, 1997.

MOHAMMADI, M. et al. Deep learning for iot big data and streaming analytics: A survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 20, n. 4, p. 2923–2960, 2018.

MOHAMMADI, M. et al. Deep learning for iot big data and streaming analytics: A survey. **IEEE Communications Surveys and Tutorials**, v. 20, n. 4, p. 2923–2960, 2018.

NEDA, N. et al. Multi-precision deep neural network acceleration on fpgas. In: **ASP-DAC**. [S.l.]: IEEE, 2022. p. 454–459.

NEVEN, D. et al. Towards end-to-end lane detection: an instance segmentation approach. In: **Intelligent Vehicles Symposium**. [S.l.]: IEEE, 2018. p. 286–291.

NING, Z. et al. Green and sustainable cloud of things: Enabling collaborative edge computing. **IEEE Communications Magazine**, IEEE, v. 57, n. 1, p. 72–78, 2018.

NURVITADHI, E. et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In: **FPGA**. [S.l.]: ACM, 2017. p. 5–14.

ONNX. **Open Neural Network Exchange**. 2023. <https://github.com/onnx/onnx>.

PACHECO, R. G.; COUTO, R. S.; SIMEONE, O. Calibration-aided edge inference offloading via adaptive model partitioning of deep neural networks. In: IEEE. **ICC 2021-IEEE International Conference on Communications**. [S.l.], 2021. p. 1–6.

PACHECO, R. G.; COUTO, R. S.; SIMEONE, O. On the impact of deep neural network calibration on adaptive edge offloading for image classification. **Journal of Network and Computer Applications**, Elsevier, p. 103679, 2023.

PANDA, P.; SENGUPTA, A.; ROY, K. Conditional deep learning for energy-efficient and enhanced pattern recognition. In: **DATE**. [S.l.]: IEEE, 2016. p. 475–480.

PANDA, P.; SENGUPTA, A.; ROY, K. Energy-efficient and improved image recognition with conditional deep learning. **ACM J. Emerg. Technol. Comput. Syst.**, v. 13, n. 3, p. 33:1–33:21, 2017.

PAPPALARDO, A. **Xilinx/brevitas**. [S.l.]: Zenodo, 2023. <https://doi.org/10.5281/zenodo.3333552>.

PASZKE, A. et al. Pytorch: An imperative style, high-performance deep learning library. In: **NeurIPS**. [S.l.: s.n.], 2019. p. 8024–8035.

PENG, H. et al. Collaborative channel pruning for deep networks. In: **ICML**. [S.l.]: PMLR, 2019. (Proceedings of Machine Learning Research, v. 97), p. 5113–5122.

POUCHET, L.-N. et al. Polyhedral-based data reuse optimization for configurable computing. In: **Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2013. p. 29–38.

RAN, X. et al. Deepdecision: A mobile deep learning framework for edge video analytics. In: IEEE. **IEEE INFOCOM 2018-IEEE conference on computer communications**. [S.l.], 2018. p. 1421–1429.

RAN, X. et al. Deepdecision: A mobile deep learning framework for edge video analytics. In: IEEE. **IEEE INFOCOM 2018-IEEE conference on computer communications**. [S.l.], 2018. p. 1421–1429.

RASTEGARI, M. et al. Xnor-net: Imagenet classification using binary convolutional neural networks. In: SPRINGER. **Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV**. [S.l.], 2016. p. 525–542.

REDDI, V. J. et al. Mlperf inference benchmark. In: **ISCA**. [S.l.]: IEEE, 2020. p. 446–459.

RUIZ, A.; VERBEEK, J. Adaptative inference cost with convolutional neural mixture models. In: **ICCV**. [S.l.]: IEEE, 2019. p. 1872–1881.

RUSCI, M.; CAPOTONDI, A.; BENINI, L. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. **Proceedings of Machine Learning and Systems**, v. 2, p. 326–335, 2020.

RUSSAKOVSKY, O. et al. ImageNet Large Scale Visual Recognition Challenge. **IJCV**, v. 115, n. 3, p. 211–252, 2015.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM Journal of Research and Development**, v. 44, n. 1.2, p. 206–226, 2000.

SANDLER, M. et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2018. p. 4510–4520.

SCHAFER, B. C.; TAKENAKA, T.; WAKABAYASHI, K. Adaptive simulated annealer for high level synthesis design space exploration. In: IEEE. **2009 International Symposium on VLSI Design, Automation and Test**. [S.l.], 2009. p. 106–109.

SCHAFER, B. C.; WAKABAYASHI, K. Design space exploration acceleration through operation clustering. **IEEE transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 29, n. 1, p. 153–157, 2009.

SCHAFER, B. C.; WAKABAYASHI, K. Divide and conquer high-level synthesis design space exploration. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM New York, NY, USA, v. 17, n. 3, p. 1–19, 2012.

SEIFEDDINE, W.; ADJIH, C.; ACHIR, N. Dynamic hierarchical neural network offloading in iot edge networks. In: IEEE. **2021 10th IFIP International Conference on Performance Evaluation and Modeling in Wireless and Wired Networks (PEMWN)**. [S.l.], 2021. p. 1–6.

SHARMA, H. et al. Dnnweaver: From high-level deep network models to fpga acceleration. In: **the Workshop on Cognitive Architectures**. [S.l.: s.n.], 2016.

SHEN, Y.; FERDMAN, M.; MILDER, P. Maximizing cnn accelerator efficiency through resource partitioning. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 45, n. 2, p. 535–547, 2017.

SHUAI, Y. et al. Memtv: a research on multi-level edge computing model for traffic video processing. In: **CAC**. [S.l.: s.n.], 2020.

SILVANO, C. et al. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. **arXiv preprint arXiv:2306.15552**, 2023.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. In: **ICLR**. [S.l.: s.n.], 2015.

SOHRABIZADEH, A.; WANG, J.; CONG, J. End-to-end optimization of deep learning applications. In: **Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2020. p. 133–139.

STMicroelectronics. **X-Cube-AI: AI Expansion Pack for STM32CubeMX**. 2019. <https://www.st.com/en/embedded-software/x-cube-ai.html>.

SUI, X. et al. A hardware-friendly high-precision cnn pruning method and its fpga implementation. **Sensors**, MDPI, v. 23, n. 2, p. 824, 2023.

SUN, F. et al. A high-performance accelerator for large-scale convolutional neural networks. In: IEEE. **2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)**. [S.l.], 2017. p. 622–629.

Synective. **Machine Learning**. 2023. <https://synective.se/services/machine-learning/>.

SZE, V. et al. **Efficient Processing of Deep Neural Networks**. [S.l.]: Morgan & Claypool Publishers, 2020. (Synthesis Lectures on Computer Architecture).

TEERAPITTAYANON, S.; MCDANEL, B.; KUNG, H. T. Branchynet: Fast inference via early exiting from deep neural networks. In: **ICPR**. [S.l.]: IEEE, 2016. p. 2464–2469.

TERMRITTHIKUN, C. et al. Eeea-net: An early exit evolutionary neural architecture search. **Eng. Appl. Artif. Intell.**, v. 104, p. 104397, 2021.

THOMPSON, N. C. et al. The computational limits of deep learning. **arXiv preprint arXiv:2007.05558**, 2020.

THOMPSON, N. C. et al. Deep learning's diminishing returns: The cost of improvement is becoming unsustainable. **ieee Spectrum**, IEEE, v. 58, n. 10, p. 50–55, 2021.

ULLAH, S.; MURTHY, S. S.; KUMAR, A. *SMApproxlib*: library of fpga-based approximate multipliers. In: **DAC**. [S.l.]: ACM, 2018. p. 157:1–157:6.

UMUROGLU, Y. et al. Logicnets: Co-designed neural networks and circuits for extreme-throughput applications. In: **Proceedings of the International Conference on Field-Programmable Logic and Applications**. Los Alamitos, CA, USA: IEEE Computer Society, 2020. p. 291–297.

UMUROGLU, Y.; JAHRE, M. Streamlined deployment for quantized neural networks. **arXiv preprint arXiv:1709.04060**, 2017.

VANHOUCKE, V.; SENIOR, A.; MAO, M. Z. Improving the speed of neural networks on cpus. **Deep Learning and Unsupervised Feature Learning Workshop, NIPS**, 2011.

VEIT, A.; BELONGIE, S. J. Convolutional networks with adaptive inference graphs. In: **ECCV (1)**. [S.l.]: Springer, 2018. (Lecture Notes in Computer Science, v. 11205), p. 3–18.

VENIERIS, S. I.; BOUGANIS, C. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In: **FCCM**. [S.l.: s.n.], 2016. p. 40–47.

Veripool. **Verilator**. 2021. <veripool.org/verilator/>.

VÉSTIAS, M. Efficient design of pruned convolutional neural networks on fpga. **Journal of Signal Processing Systems**, Springer, v. 93, n. 5, p. 531–544, 2021.

WANG, M. et al. Dynexit: A dynamic early-exit strategy for deep residual networks. In: **SiPS**. [S.l.]: IEEE, 2019. p. 178–183.

WANG, X. et al. Convergence of edge computing and deep learning: A comprehensive survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 22, n. 2, p. 869–904, 2020.

WANG, X. et al. Skipnet: Learning dynamic routing in convolutional networks. In: **Proceedings of the European conference on computer vision (ECCV)**. [S.l.: s.n.], 2018. p. 409–424.

WANG, Y. et al. Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In: **Proceedings of the 53rd Annual Design Automation Conference**. [S.l.: s.n.], 2016. p. 1–6.

WANG, Z. et al. Approximate multiply-accumulate array for convolutional neural networks on FPGA. In: **ReCoSoC**. [S.l.]: IEEE, 2019. p. 35–42.

WU, H. et al. Integer quantization for deep learning inference: Principles and empirical evaluation. **CoRR**, abs/2004.09602, 2020.

Xilinx Inc. **7 Series FPGAs Configurable Logic Block**. 2016. <https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf>.

Xilinx Inc. **A HLS-based Deep Neural Network Accelerator library for Xilinx Ultrascale+ MPSoC devices**. 2020. <github.com/Xilinx/CHaiDNN>.

Xilinx Inc. **Vitis AI**. 2023. <https://xilinx.github.io/Vitis-AI>.

XU, Z. et al. Reform: Static and dynamic resource-aware DNN reconfiguration framework for mobile device. In: **DAC**. [S.l.]: ACM, 2019.

XU, Z. et al. Directx: Dynamic resource-aware cnn reconfiguration framework for real-time mobile applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 40, n. 2, p. 246–259, 2020.

YANG, L. et al. Resolution adaptive networks for efficient inference. In: **CVPR**. [S.l.]: Computer Vision Foundation / IEEE, 2020. p. 2366–2375.

YANG, T. et al. Netadapt: Platform-aware neural network adaptation for mobile applications. In: **ECCV (10)**. [S.l.]: Springer, 2018. (Lecture Notes in Computer Science, v. 11214), p. 289–304.

YOU, W.; WU, C. Rsnn: A software/hardware co-optimized framework for sparse convolutional neural networks on fpgas. **IEEE Access**, IEEE, v. 9, p. 949–960, 2020.

YOUSSEF, E. et al. Energy adaptive convolution neural network using dynamic partial reconfiguration. In: **MWSCAS**. [S.l.]: IEEE, 2020. p. 325–328.

YU, J.; HUANG, T. Autoslim: Towards one-shot architecture search for channel numbers. **arXiv preprint arXiv:1903.11728**, 2019.

YU, J. et al. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In: **ISCA**. [S.l.]: ACM, 2017. p. 548–560.

ZHANG, C. et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In: **FPGA**. [S.l.: s.n.], 2015. p. 161–170.

ZHANG, T. et al. The design and implementation of a wireless video surveillance system. In: **MobiCom**. [S.l.]: ACM, 2015. p. 426–438.

ZHAO, Z. et al. Edgeml: An automl framework for real-time deep learning on the edge. In: **IoTDI**. [S.l.]: ACM, 2021. p. 133–144.

APPENDIX A — PRUNING AND HLS ON THE CHAIDNN ACCELERATOR

This appendix presents the first work carried out in this thesis. It explored pruning and HLS for optimizing CNN acceleration on FPGAs. However, it was based on the CHaiDNN accelerator, which was discontinued by Xilinx and, because of that, was not taken into our further workfronts. Nevertheless, this work follows the general workfront structure presented in Figure 4.1.

This work started with the observation that, while a few works have already proposed FPGAs accelerators capable of running pruned CNN models and others have exploited HLS to generate different acceleration versions, they have not yet been considered altogether (which would multiply the design options). Most importantly, such works are restricted to a static design exploration, not adapting to the changes the system needs. In this work we approached the adaptability problem by exploiting pruning and HLS on quantized CNNs. We called this AdaServ for Adaptive Serving of inferences, presented next.

A.1 AdaServ

Figure A.1 – Workflow for the AdaServ.

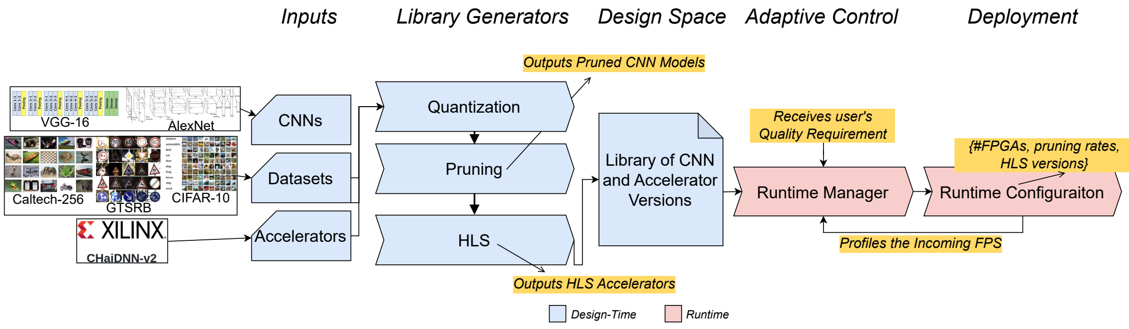


Figure A.1 details the AdaServ workflow. Different from the thesis main workflow (from Figure 4.1), AdaServ does the following:

- AdaServ uses only the pruning and HLS optimizations on quantized CNNs (therefore, only the three optimizations are "in use" as Library Generators in Figure A.1);
- AdaServ was evaluated with the single-engine CHaiDNN accelerators and its sup-

ported CNN/datasets[1];

- the Runtime Manager that besides the system profiling (i.e., incoming Frames per Second), also receives a configurable Quality Requirement from the user (also highlighted in Figure A.1);
- AdaServ's Runtime Configuration defines the current number of allocated FPGAs (evaluated from one to four) and the level of the pruning and HLS optimizations.

More specifically, the Library Generators step in AdaServ (i.e., design-time phase in Figure A.1) generates Pruned CNN Models and the HLS accelerator versions. Pruning was used in such a way to generate *Pruned CNN Models*, which are multiple variants of popular CNN models produced with different pruning rates. Each of the generated pruned CNN models provides a different trade-off between model size and accuracy. At the hardware level, on the other hand, HLS was used to generate *HLS Accelerators* with variations in the HLS pragmas in charge of the their engines to generate versions of the same HLS behavioral description. These accelerators vary in processing capabilities, FPGA resource usage, and power profiles. Hereafter, the HLS generated accelerators will be identified with a version number. It is also important to note that, because single-engine accelerators (Subsection 2.2.2.1) are generic and execute in a layer-by-layer fashion, accelerators and pruned CNN models are decoupled (i.e., accelerators are synthesized independently from the CNNs) in this workfront. So, any accelerator can execute any CNN model of any pruning rate.

The Runtime Manager performs the Adaptive Control. It searches for the best Runtime Configuration (which, in AdaServ, defines the pruned CNN models at particular rates, the HLS generated accelerator versions, and the number of allocated FPGAs). The Runtime Manager in AdaServ also performs workload profiling and takes a global Quality Requirement (see Figure A.1) that will define the minimum accuracy to process inferences. Next, we provide in-depth details of AdaServ's steps of Library Generators (design-time) and Adaptive Control (runtime).

A.1.1 Library Generators

Pruning Optimization. We call Uniform Pruning the algorithm developed to generate AdaServ's pruned CNN models. It leverages the filter selection from (LI et al.,

[1]We have also evaluated AdaServ with a higher resolution dataset (reported in Appendix C).

2017) and was implemented on top of PyTorch (PASZKE et al., 2019), an open-source machine learning framework. As shown in Algorithm 3, AdaServ uniformly removes channels from all CONV layers of the CNN, based on a given pruning rate. After the process of pruning and retraining for a particular pruning rate finishes (Line 14 in Algorithm 3), the correspondent pruned CNN model is saved to the library along with information on its test accuracy (from the evaluation in Line 15), which will be used during the online phase. AdaServ repeats this process for different pruning rates.

Algorithm 3 AdaServ's Uniform Pruning

```
1: Set the initial pruning rate $pr$
2: Set the pruning rate step $\alpha$
3: Set the number of retraining epochs $re$
4: for For each dataset $ds$ do
5:     for For each original CNN model $m$ do
6:         while $pr < 100\%$ do
7:             for For each convolutional layer $i$ do
8:                 Find the number of channels to remove $r_i$ based on $pr$
9:                 for For each filter $f_{i,j}$ of $i$ do
10:                    Calculate the sum of its absolute kernel weights $s_j = \sum_{l \in |kernels\ of\ i|} \sum |K_l|$
11:                Sort filters by $s_j$
12:                Prune $r_i$ filters with the lowest $s_j$ values from $i$ and remove the corresponding kernels of layer $i+1$
13:                Retrain the pruned CNN model for $re$ epochs
14:                Evaluate the pruned CNN model on $ds$ and save its accuracy $acc$
15:                Increase $pr$ by $\alpha$
16:                Append the pruned CNN model to library along with its $acc$ and $pr$
17:        Reset $pr$ to $0\%$
```

HLS Optimization. For AdaServ, we explore HLS with the CHaiDNN-V2 in two different ways:

- Implementing the accelerator with a different number of convolution engines, so it can process convolutional layers that have no data-dependency (i.e., layers processing different input images) in parallel. Due to resource limitations of the FPGA in use, mainly the AXI interface, AdaServ generates accelerators with up to three convolution engines.
- Changing the internal convolution engine parallelism (expressed mainly in terms of the DSP blocks implementing the MACs), which is done by tuning the UNROLL pragmas in charge of processing the convolutional layers.

With these variations, each HLS accelerator version presents a particular resource usage, latency, and power profiles. After synthesized into an FPGA bitstream, each accelerator version is saved in the library (following the workflow in Figure 4.1) with their reported latency (obtained from automatic RTL simulations) and power dissipation (obtained through synthesis).

To give a concrete example of the varied profiles of the HLS generated accelerators, let us compare two versions from our experiments. The first one is an accelerator version that gives low latency, it is synthesized using 1418 parallel DSPs with a single convolution engine. The second one is an accelerator of high throughput that is synthesized using 624 DSPs and three convolution engines (208 DSPs each). Considering those examples, we have the first one processing inferences at latency, on average, 1.96x lower than the second one. In contrast, the second version, with three parallel engines, increases the average throughput in 1.53x over the low-latency version. Besides FPGA resource usage and performance, the accelerator versions also play a significant role in the FPGA power dissipation. For example, with the accelerators used in our experiments, power dissipation ranges from 6.282 to 16.848 Watts.

In summary, the CNN models and the accelerators generated are stored in the library in the form of a table containing a list of pruned CNN models (rows) with their accuracy as well as the throughput and power values when executing on each accelerator (columns) - 'Library of CNN and Accelerator Versions' in the design-time step of Figure A.1. Next, we discuss how AdaServ adapts to changes in workload and quality requirements based on the created library.

A.1.2 Adaptive Control

Runtime Manager. At runtime, the Runtime Manager is the software module (running on the FPGA co-processor, a Quad-Core ARM Cortex-A53) responsible for, besides the search for Runtime Configurations, sending the inference requests (frames) to be inferred by the pruned CNN model and accelerator version specified by the configuration; and, 3) profiling the server workload (based on the incoming FPS - see Figure A.1). Whenever it identifies a change in workload, an interruption is sent, forcing a new search. Additionally, if the new configuration differs from the current one, the Runtime Manager can reconfigure an already in use FPGA with a different accelerator or allocate a new FPGA.

Broadly speaking, the main objective of the Runtime Manager is to find configurations that maintain the accuracy at a certain pre-defined degree (user's Quality Requirement) while optimizing the use of resources. Every time that the Runtime Manager flags a change in workload or Quality Requirement, a new search for a configuration takes place. Considering a single FPGA board, the number of possible configurations is given by the number of pruned CNN models and HLS accelerator versions. For example, in our case study, there are 42 pruned models for each dataset (21 from the original AlexNet model and 21 from VGG-16) and four accelerator versions, giving 168 possible combinations for a single FPGA board. However, we have four FPGA boards connected, totaling 168^4 possible combinations. Thus, an exhaustive search is unfeasible due to (*i*) the large number of possible combinations and (*ii*) the necessity of re-evaluating the configuration every time there is a shift in workload (e.g., demanding more throughput from the server) or a change in Quality Requirement (e.g., lowering the accuracy threshold, allowing pruned models of higher pruned rate).

Algorithm 4 Automatic Selection

```
 1: Set the workload $FPS$, in frames per second, profiled at runtime
 2: Set the quality requirement $Q$ to the value informed by the user
 3: Set $n$, the number of allocated FPGAs, to 0
 4: Set $FPS_{config}$, the theoretical throughput of the current configuration, to 0
 5: Set $Configs$, the list of configurations per allocated FPGA, to an empty list
 6: Load the HLS accelerators power and latency reports from the library
 7: Load the pruned CNN models accuracy reports from the library
 8: while $FPS_{config}$ is below $FPS$ do
 9:     if there are still FPGAs available then
10:         Increment $n$ by one
11:     else
12:         Return $Configs$ as the configuration with $n$ allocated FPGAs (potential frame loss)
13:     Let $Candidates$ be a list with all pruned CNN models with accuracy above $Q$ on each dataset with each accelerator version
14:     Set $Gap\_FPS$ as $FPS - FPS_{config}$
15:     if there are no configurations with throughput below $Gap\_FPS$ in $Candidates$ then
16:         Store in $c$ the accelerator version and the datasets' pruned CNN models of the configuration of highest throughput
17:     else
18:         From the set of configurations with FPS above $Gap\_FPS$, store in $c$ the accelerator version and the datasets' pruned
    CNN models of the configuration with lowest power dissipation
19:     Increase $FPS_{config}$ with the throughput provided by $c$
20:     Append $c$ to $Configs$
21: Return $Configs$ as the configuration with $n$ allocated FPGAs
```

For searching such a large number of configurations, let us present the fully automatic and adaptive algorithm running in Adaserv (detailed in Algorithm 4). It requires no need for any predefined user configuration. Algorithm 4 breaks the search space into two smaller problems. At a lower level, it starts by selecting a configuration (pruned CNN model and accelerator version pair) for a particular FPGA through the following steps. First, it filters out pruned CNN models with accuracy below the current Quality Requirement (line 14). After that, Algorithm 4 picks the configuration that delivers the highest throughput (line 17). The only exception happens when the desired FPS level (profiled at runtime) can be matched by other configurations besides the one with the highest throughput (*else* condition in line 18). In that case, the configuration that dissipates the least power will be selected (line 19). In summary, the search for configuring one FPGA is exhaustive at the lower level. However, at the higher level, it greedily allocates FPGAs without considering all boards simultaneously (while the desired FPS is not matched, line 8). In this way, we guarantee a solution with the highest throughput for the given Quality Requirement, but that may not provide the best throughput-power trade-off globally (across all FPGAs). Nevertheless, we show in the following results section that this solution is fast because of the considerable search space reduction; and scalable because it makes it possible to add new CNN models (with multiple pruned versions) and new HLS accelerator versions without incurring in significant overheads since it affects only the local search (consisting of finding the pruned CNN model and accelerator version pair that gives the highest throughput for a *single* FPGA under *current* workload and Quality Requirement). We call AdaServ running with Algorithm 4 the "Automatic AdaServ." We have also implemented a user-controllable search, which we call User-Configurable AdaServ. It sorts configurations based on a user-defined weighted search (that considers the configurations' throughput and power). For the sake of simplicity, we placed its detailed explanation and its evaluations in Appendix B.

After finding a feasible Runtime Configuration through the Automatic Search, the Runtime Manager reconfigures (if necessary) the FPGAs and sends a signal the resume of the inference processing.

A.2 Results

A.2.1 Evaluation Scenario

Setup and Tools. Our simulation setup consists of a set of four Xilinx Zynq Ultrascale+ MPSoC ZCU102 boards interconnected over Ethernet, where each board can be reconfigured with one version of the HLS generated accelerators. Accelerators versions used across our experiments were synthesized from the CHaiDNN-v2's HLS (Xilinx Inc, 2020) using the Xilinx SDx 2018.3 at 300MHz. For this work, we used the CHaiDNN-v2 accelerator on ZCU102 boards (one to four interconnected over Ethernet). The CHaiDNN-v2 versions synthesized are presented in Table A.1, where #Engines represents the number of the convolution engines inside each accelerator. We used Xilinx Vivado tool for power extraction and latency of the FPGA-executed parts and the cycle-accurate Gem5 simulator (BINKERT et al., 2011) for latency of the software executed parts. CNN models are automatically quantized to 8 bit fixed-point by CHaiDNN-v2 (after being converted to Caffe (JIA et al., 2014)).

We adopt two CNN models for evaluations: AlexNet and VGG-16 (supported by CHaiDNN-V2). Models were adapted to three datasets (with the same input layer size for

Table A.1 – Accelerator versions with resource usage and power dissipation (on single XCZU9EG FPGA).

Accel. Ver.	#Engines	BRAM (%)	DSP (%)	FF (%)	LUT (%)	Power (W)
1	1	82.62	56.27	33.01	49.73	16.848
2	1	20.12	8.25	10.73	15.16	6.282
3	2	39.00	17.00	20.00	29.09	8.936
4	3	57.46	24.76	29.13	42.04	11.282

all and last FC layer adapted to match the number of classes in the dataset):

- The CIFAR-10 dataset (KRIZHEVSKY; HINTON, 2009). This dataset contains 50,000 training and 10,000 test images. The classification task consists of 10 classes ranging from "airplane" to "bird," and "truck."
- The German Traffic Sign Recognition Dataset (GTSRB) (HOUBEN et al., 2013), which contains 39,209 training and 12,630 test images of 43 classes representing road signs like speed limits and stop signs.
- The Caltech-256 dataset (GRIFFIN; HOLUB; PERONA, 2007) contains 30,607 images with no training and test partitions. The dataset was, then, split into 80% for training and 20% for test. This dataset provides 257 classes ranging from "frog" to "sail boat."

AdaServ generates 21 pruned CNN models for each original CNN with pruning rates ranging from 0% to 99% (at 5% steps). Those models are deployed within the boards' SD-Cards and further loaded by the on-board software stack.

We evaluate AdaServ on a typical smart video surveillance system with end nodes producing inference requests at the real-time rate of 25 FPS on three different machine learning tasks, represented as the three distinct image classification datasets from Section 4.1. Evaluations are 500 seconds long. During a single run, we range the number of connected end nodes from 3 to 60 (three end nodes are added every 25 seconds). Each end node continuously requests inferences on one a single dataset. For example, evaluation starts with one single node requesting inferences on CIFAR-10, another on GTSRB, and the third one requesting on Caltech-256. Evaluation ends with 20 end nodes per dataset, 60 end nodes in total. Based on the evaluated CNN models, we have set two quality requirements: 75 and 90% accuracy thresholds to evaluate AdaServ's performance, energy efficiency, and Quality of Experience (QoE).

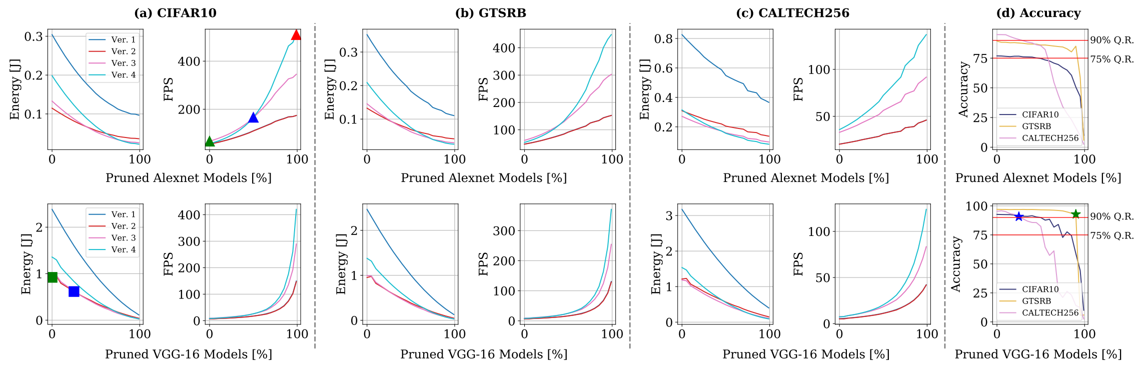
Baselines. We have selected the following baselines for comparison:

- The original Xilinx's CHaiDNN-v2 accelerator (Xilinx Inc, 2020) (accelerator version 1 in Table A.1) without HLS exploration executing the original CNN models (i.e. without pruning) as in (JIANG et al., 2018) at one, two, three, or four FPGA boards. We call **xFPGA-Alex** the baseline serving the original AlexNet, and **xFPGA-VGG16** the baseline serving the original VGG-16, where x gives the number of allocated FPGAs.

- One baseline for evaluating the **DYN**amic adaptation of **PR**uned CNN models only. It is called **DynPR** since it dynamically selects (for each inference request) the CNN model (VGG-16 or AlexNet) and pruning rate (0% to 99%) that has the lowest latency with accuracy above the quality requirement, without any changes to the original CHaiDNN-v2 accelerator from Xilinx (i.e., we do not explore HLS). DynPR is configured with four FPGAs, each running the original Xilinx's CHaiDNN-v2 accelerator. Quality requirement is indicated as y in DynPR-y.
- Same reasoning as the previous baseline, but now for evaluating the **DYN**amic adaptation of the HLS generated **AC**celerators only. We call this baseline **DynAC**. It can dynamically change the accelerator version by reconfiguring FPGAs at runtime but executing CNN models without pruning. DynAC chooses HLS accelerators of the lowest power dissipation with throughput above the current workload (given by the total rate of incoming frames for inference). DynAC uses four FPGAs and original VGG-16 (for 90% quality requirement) and AlexNet (for 75% quality requirement) models. Quality requirement is indicated as y in DynAC-y.
- A final baseline evaluating both HLS-generated accelerators and pruned models, where the **Best PR**uning and **AC**celerator is statically chosen from the library using an exhaustive search considering a given quality requirement. Therefore, this configuration does not change during execution (i.e. this represents AdaServ without adaptation capabilities). We call this baseline **BestPRAC**-x, where x gives the quality requirement. The DynPR, DynAC, and BestPRAC baselines run on four FPGAs since it gave the highest performance in our experiments.

A.2.2 Evaluation

Figure A.2 – Energy and FPS for the four accelerator versions running AlexNet (upper plots) and VGG-16 (lower plots) models on CIFAR-10 (a), GTSRB (b), and Caltech-256 (c) datasets, with their accuracy (d).



Opportunities for Optimization. Figure A.2 from (a) to (c) show the evaluated datasets. For each dataset, we present pruned AlexNet models at the top and pruned VGG-16 models at the bottom plots with pruning rates from 0 to 99% (x-axis). Energy and FPS (y-axis) are given on the four accelerator versions (Ver. 1-4) running on a single FPGA, see Table A.1 for the library's accelerator versions. Accuracy of pruned AlexNet and VGG-16 models on all datasets are presented in Fig. A.2(d).

Initially, let us consider the accuracy of pruned VGG-16 and AlexNet models in Fig. A.2(d). Pruned CNNs and accelerators will have different optimal configurations depending on the user's quality requirements and workload. Fig. A.2(d) shows that the Uniform Pruning affects the models' accuracy differently depending on the task's dataset. Both AlexNet and VGG-16 show greater resilience to pruning on the GTSRB dataset (slow fall of yellow curves in Fig. A.2(d)). That is, we see that many pruned CNN models present feasible accuracy levels (above threshold) on this dataset, enlarging the design space when serving inferences on GTSRB. For example, let us consider the "75% Q.R." and "90% Q.R." accuracy thresholds (horizontal red lines in Fig. A.2(d)). Then, even models with aggressive pruning rates, like pruned VGG-16 model of 90% pruning rate (green star in Fig. A.2(d)'s bottom plot), achieve accuracy above the "90% Q.R." threshold on GTSRB. In contrast, for the Caltech-256 dataset (pink curves in Fig. A.2(d)), we can see a steeper accuracy curve for AlexNet and VGG-16 models, reducing optimization opportunities (fewer pruned CNN models will stay above quality thresholds). For example, for pruned VGG-16 models on Caltech-256, the smallest model that stays above the

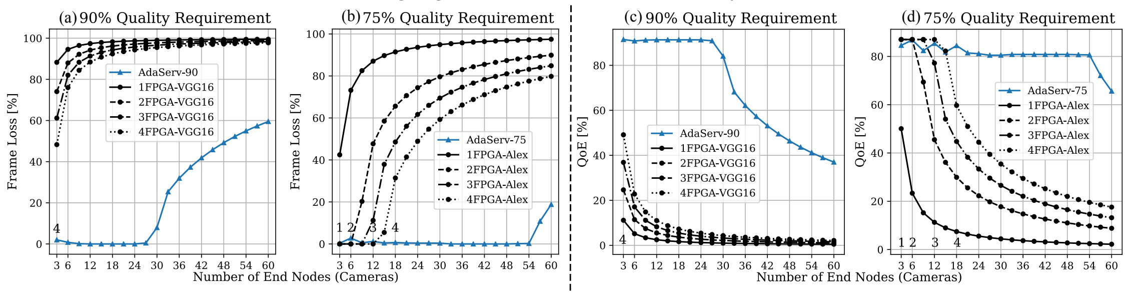
"90% Q.R." accuracy threshold is the one obtained with 25% pruning rate (blue star in Fig. A.2(d)'s lower plot). This overall behavior indicates that it is not possible to (statically) choose a single best CNN and pruning rate covering all possible quality requirements because the pruning rate affects models and datasets differently on accuracy (Fig. A.2(d)). Moreover, as we show next, pruned CNN models present varied performance depending on the accelerator executing them (Fig. A.2(a) to (c)), which indicates that the selection of HLS accelerators should also be considered when shifts in quality requirement and workload (e.g., not a constant number of connected end nodes) are expected.

As it can be seen in Fig. A.2(a)'s top plot, for pruned AlexNet, accelerator version 3 provides the highest FPS when running models from 0% (green triangle in Fig. A.2(a)) to 50% (blue triangle in Fig. A.2(a)) pruning rate on CIFAR-10; and, 0% to 40% pruning rates for GTSRB tasks (as shown in Fig. A.2(b)). HLS accelerator version 4, on the other hand, provides the highest FPS on CIFAR-10 when executing pruned CNNs for the pruning rates from 55% up to the smallest model of 99% pruning rate (red triangle in Fig. A.2(a)). To have a more concrete idea of the effects of those pruning levels, the 50% pruning rate, for example, corresponds to reductions of about 75% in the amount of memory transfers and MAC operations in the case of VGG-16 and 72% in the case of AlexNet. For more aggressive pruning rates, such as 95%, reductions of over 99% in memory transfers and MAC operations are achieved. Nevertheless, we recall that aggressive pruning rates come with significant costs in accuracy as shown in Fig. A.2(d) and, as such, pruned CNNs cannot be naively selected from a performance-only perspective.

The heterogeneity observed in the performance delivered by the accelerator versions and pruned CNN models also holds for energy. For example, we have accelerator version 2 consuming the least energy on CIFAR-10 when executing pruned VGG-16 models of 0 to 25% pruning rates (green and blue squares, respectively, in Fig. A.2(a)'s bottom plot). Accelerator version 3, on the other hand, consumes the least energy for VGG-16 pruned models of 0 to 70% pruning rates on Caltech-256 and 25 to 85% on GTSRB. As can be observed, a similar behavior also holds for pruned AlexNet models (top plots in Fig. A.2 (a), (b), and (c)). This energy and performance heterogeneity combined with the unpredictability of workload and quality requirements means that static systems cannot achieve optimal efficiency: online adaptation of pruning and accelerators is essential to accommodate the levels of unpredictability faced by modern CNN applications. Next, we discuss how AdaServ's dynamic selection takes advantage of this design space for effectively processing inference at scale.

AdaServ vs. Original Baselines. Based on the aforementioned discussion, we compare AdaServ to state-of-the-art baselines that deploy original CNN models (without pruning) on original HLS accelerators without any optimization.

Figure A.3 – Frame Loss and Quality of Experience (QoE) for 90% (plots a and c) and 75% (plots b and d) quality requirements for AdaServ and original baselines. Changes in AdaServ's number of allocated FPGAs are indicated (where not indicated, the number of allocated FPGAs remains as the last indicated change - next to the left). X-axis indicate the number of connected cameras changing over time (3 added every 25 seconds).



We evaluate AdaServ's performance in terms of the Edge's loss of inference requests (frame loss in our use case) against the state-of-the-art baselines xFPGA-VGG16 and xFPGA-Alex, where x indicates their number of allocated FPGAs. xFPGA-VGG16 and xFPGA-Alex deploy original (not pruned) VGG-16 and AlexNet models on the original Xilinx's CHaiDNN-V2 accelerator (version 1 in Table A.1). Fig. A.3 presents the results on frame loss for both quality requirements: 90% in Fig. A.3(a) and 75% in Fig. A.3(b).

90% Quality Requirement: Due to the higher quality requirement imposed by this scenario, AdaServ selects only pruned VGG-16 models for CIFAR-10 inferences, but may select pruned AlexNet models for GTSRB (up to 85% pruning rate) and Caltech-256 (up to 55% pruning rate). We can see that the xFPGA-VGG16 baseline configurations (black curves in Fig. A.3(a)) cannot keep up with real-time inference even for the lightest workload (one camera requesting inference on each task, three cameras in total). For instance, 4FPGA-VGG16 (that uses 4 FPGAs), serving only three end nodes, loses 48.28% of the requested frames. In contrast, AdaServ stays below the 60% frame loss mark at any point during execution, even for the full workload of 60 end nodes, when it loses 59.51% of the frames. In Fig. A.3(a), we see AdaServ's online selection of pruned CNN model and accelerator versions drastically reducing frame losses for the 90% quality requirement (AdaServ-90), enabling a reasonably low level of missed inferences considering the extremely high workload.

On the 90% quality requirement, AdaServ executed three different configurations over the full 500 seconds run. Due to the high quality requirement, it is only possible to select large and compute-intensive CNN models (e.g., lightly pruned VGG-16 models - the initialization of such large models caused the initial peak in frame loss for three end nodes in A.3(a)), so AdaServ could not allocate less than four FPGAs. Still, AdaServ could save some power by allocating accelerator versions of low resource usage in configurations for workloads of three and six end nodes with no performance loss (while original CHaiDNN-v2 accelerators on one to four FPGAs were lose at least 48% of the requested inferences). Over the full run, two FPGA reconfigurations were performed and configurations covered seven different pruned CNN models and two HLS accelerator versions. With this quality requirement, AdaServ was capable of scaling to up to 30 end nodes without significant performance degradation (below 7.95% frame loss). In terms of accuracy over the full run, AdaServ showed an average accuracy of 90.47% (which is 2.18% below the baselines running original VGG-16 models) regarding the CIFAR-10 dataset. As for the other two datasets, AdaServ provided an average accuracy of 92.97% (4.04% below VGG-16 baselines) on GTSRB, and 90.86% on Caltech-256 (4.44% below VGG-16 baselines). Nevertheless, the loss in accuracy comes with an average frame loss of 23.34%, while the best performing original CHaiDNN-V2 deploying original VGG-16 models (4FPGA-VGG16) ends up with an average frame loss of 91.42%.

75% Quality Requirement: Considering the lower accuracy threshold of this scenario, we compare AdaServ to the original CHaiDNN accelerators executing original AlexNet models for inference (1, 2, 3, and 4FPGA-Alex). Still, AdaServ may also deploy the high-accuracy pruned VGG-16 models (whenever possible) for inference since it can dynamically adapt its configuration. Under that quality requirement, AdaServ was capable of finding configurations that kept the frame loss below 18.82% at any moment. AdaServ allocated one FPGA for serving workloads of three end nodes, two FPGAs for up to 12, and three FPGAs for up to 18 end nodes. The fourth FPGA was only allocated for workloads of 18 end nodes or higher (number of allocated FPGAs are indicated in Fig. A.3(b)). Contrarily, 4FPGA-Alex (the best performing original CHaiDNN accelerator deploying AlexNet CNN Model) could only achieve similar levels of frame loss for up to 15 end nodes. Considering accuracy, AdaServ finished the evaluation with an average accuracy of 76.34% (that is only 0.56% below the baselines running original AlexNet models, without pruning) on the CIFAR-10 dataset, and 87.23% (2.08% below baseline) and 82.79% (11.84% below baseline) on GTSRB and Caltech-256 datasets, respectively.

On the lower quality requirement (75%), AdaServ used 10 different configurations across the 500 seconds run. AdaServ started with one FPGA configured with the HLS accelerator version 3 executing VGG-16 at 80% pruning rate for GTSRB and CIFAR-10 inferences and AlexNet at 55% for Caltech-256 inferences. Then, when the workload increases to six cameras, AdaServ allocates a new FPGA with HLS accelerator version 1. Changes of accelerator versions and pruned CNN models continue until the 30th node is connected when AdaServ has all four FPGAs configured with the HLS accelerator version 4 running pruned AlexNet models of 90%, 55%, and 45% pruning rates for the GTSRB, Caltech-256, and CIFAR-10 datasets, respectively. This last configuration lasts until the end of the execution. Overall, AdaServ reconfigured FPGAs five times using three different HLS accelerator versions and 16 different pruned CNN models. Changing between configurations causes frame losses that are equivalent to 37 frames due to FPGA reconfiguration, about 90ms each and, and 51 frames due to loading pruned CNN models over the full run, totalling 0.15% idle-time over the full 500 seconds run. Particularly, the peak in frame loss when six end nodes are connected is caused by reconfiguring the only FPGA in use, while the second one was being added. The initial Runtime Manager search takes about 53ms (subsequent searches do not require stopping the inference processing).

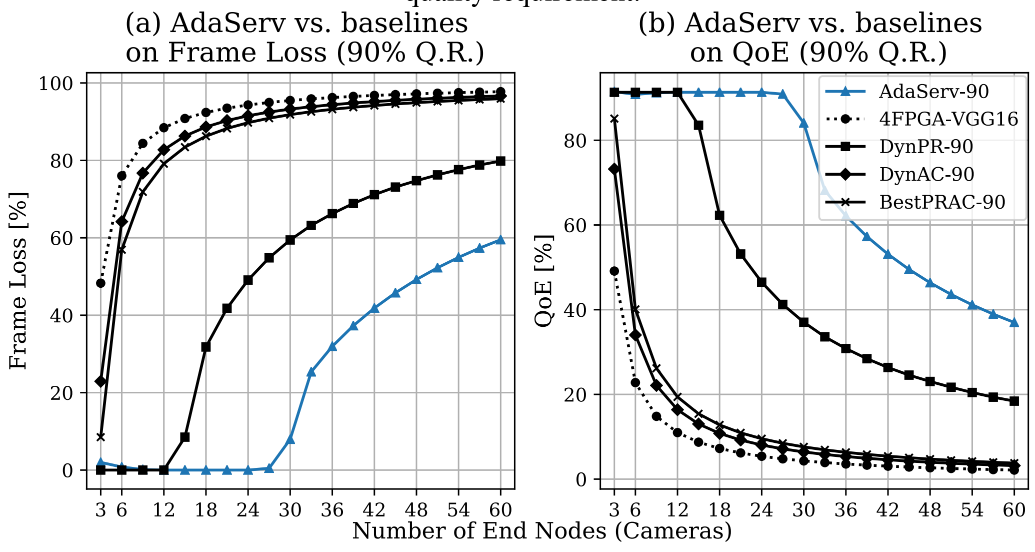
In summary, our 2-step approach provided the best performance with a smart resource allocation. Two reasons explain this: first, the offline phase, which takes advantage of pruned CNN models and accelerator versions to enable a vast design space; and, second, the Runtime Manager's online profiling that monitors incoming FPS and signals for adapting the inference processing. Next, we evaluate AdaServ in terms of Quality of Experience (QoE) to assess the inference trade-off between performance and accuracy.

Regarding Quality of Experience (QoE), Fig. A.3(c) and (d) show QoE for both executions under 90% (a) and 75% (b) quality requirements. Contrarily to the rapid fall of QoE of 1/2/3/4FPGA-VGG16 baselines, AdaServ could exchange some of the original VGG-16's accuracy for performance by allocating pruned models on four FPGA boards on the 90% quality requirement.

Under the 75% quality requirement, AdaServ's Runtime Manager selects pruned VGG-16 models for GTSRB and CIFAR-10 inference with up to nine end nodes. However, the higher accuracy and slower inference of pruned VGG-16 models (in contrast with AlexNet models, see Accuracy in Fig. A.2(d)) produces levels of QoE lower than the baselines only for relatively low workloads (up to 15 end nodes). Then, the increase in workload (and consequent frame losses) forces AdaServ to switch to pruned AlexNet models that have faster inference. Nevertheless, it is worth mention that during that interval, from 3 to 15 end nodes, AdaServ kept its scalability and efficiency goals by using at most three FPGAs. Only with 18 end nodes, AdaServ finds configurations consisting of pruned AlexNet models for requests on all datasets. From that point until the end of execution, AdaServ shows a QoE higher than all baseline configurations. It is also noticeable peaks in QoE, like from 15 to 18 end nodes (from 81.79% to 84.48% QoE). At that point in time, AdaServ allocates the fourth FPGA board.

In the end, AdaServ achieves a good level of compromise between accuracy and performance (QoE) that is higher than baselines for most of the evaluations. AdaServ assures that the accuracy degradation, due to the pruned CNN models, is always kept above the user's quality requirement by the Runtime Manager at high-performance levels.

Figure A.4 – Frame loss (a) and QoE (b) of dynamic accel. selection (DynAC), dynamic pruned CNN (DynPR), and best static configuration (BestPRAC) baselines against AdaServ on the 90% quality requirement.



AdaServ vs. dynamic pruning (DynPR), dynamic accelerator (DynAC), and best static pruning/accelerator (BestPRAC) baselines. To show that dynamically adapting HLS accelerator version and pruned CNN models synergistically is essential, we compare AdaServ with three optimized baselines: DynPR only adapting the pruned CNN model but using the original CHaiDNN-v2 HLS accelerator; the DynAC that does the opposite: it fixes the CNN models to their original sizes (0% pruning rate), but exploits the library' HLS accelerator versions whenever it needs to increase throughput at the cost of increased power dissipation; and the BestPRAC, which is statically configured with the best accelerator version and the best pruned CNN model, without any changes during execution. Recall that these baselines run over four FPGA boards and have the quality requirement denoted as y in DynPR-y, DynAC-y, and BestPRAC-y. We present Frame Loss (Fig. A.4(a)) and QoE (Fig. A.4(b)) of AdaServ against the aforementioned baselines on the 90% quality requirement (75% presents similar behavior). The best static baseline from previous subsection, which uses only the original accelerator without model pruning (4FPGA-VGG16), is also shown for comparison.

First, we note that all optimized baselines already increase the number of served inferences and QoE over the 4FPGA-VGG16 baseline (Fig. A.4). For example, on the 90% quality requirement, DynPR-90, DynAC-90, and BestPRAC-90 achieve average frame losses of 42.65, 4.22, and 6.8% below 4FPGA-VGG16 on the same quality requirement. The same behavior holds for the 75% quality requirement. Therefore, we have that dynamically exploring even part of the design space (DynAC adapting HLS accelerators or DynPR adapting pruned CNN models) or statically configuring both HLS and CNN is already beneficial to serve inferences at performance levels and scalability higher than state-of-the-art baselines.

Second, we point out that a static exploitation of the design space on the two optimization axis (CNNs and HLS accelerators) is not as beneficial as dynamic adaptation of pruned CNN models. That is, the DynPR baseline achieves better frame loss and QoE rates than the static configuration of accelerator version and pruned CNN model provided in BestPRAC on the 90% quality requirement. For example, BestPRAC-90 achieves average frame loss of 35.82% worse than DynPR-90. Moreover, DynPR also shows the best scalability among the evaluated baselines: no significant frame losses up to 12 end nodes on the 90% quality requirement and up to 27 end nodes for the 75% quality requirement (not pictured in Fig. A.4). Despite the improvement over 4FPGA-VGG16, DynAC, and BestPRAC baselines, DynPR cannot achieve performance and scalability levels of our two-step approach that dynamically adapts both pruned CNN models and HLS accelerator. For example, AdaServ-90 more than doubles the number of cameras that can be served without significant losses from 12 in DynPR-90 to 27. Fig. A.4(a) also shows that AdaServ's frame loss stays below DynPR's during all execution. Therefore, it becomes clear that is not enough to simply enable a dynamic system that either (i) adapts the CNN models ignoring their impact on hardware (DynPR); or (ii) adapts the HLS accelerators running non-optimized CNN models (DynAC); or, even, (iii) statically chooses HLS accelerators and pruned CNN models from the design space (BestPRAC). It is required to dynamic adapt the HLS accelerators at runtime as well as the pruned CNN models to cope with the constant workload changes and unpredictable quality requirements.

Table A.2 – Power dissipation of AdaServ and baseline configurations.

	AdaServ-75	AdaServ-90	1FPGA-VGG16/Alex	2FPGA-VGG16/Alex	3FPGA-VGG16/Alex	4FPGA-VGG16/Alex
Power (W)	33.14	35.74	16.85	33.7	50.54	67.40
	DynPR-75	DynPR-90	DynAC-75	DynAC-90	BestPRAC-75	BestPRAC-90
Power (W)	67.40	67.40	61.08	61.83	45.13	45.13

Besides gains in performance and QoE, dynamically adapting pruning and accelerators altogether is also more efficient. As mentioned earlier, our approach allocates new FPGAs only when demanded. Table A.2 shows the average power dissipation of AdaServ and baselines (under 75% and 90%). xFPGA-Alex and xFPGA-VGG16, where x is the number of FPGAs, that use the original CHaiDNN accelerator are shown as xFPGA-VGG16/Alex since they present the same power dissipation (same FPGA configurations). It is clear that AdaServ can attend a greater number of inferences while keeping an intermediate power dissipation thanks to the smart selection of pruned CNNs, HLS accelerator versions, and number of FPGAs. Gains in power dissipation and the higher number of processed inferences grant AdaServ energy efficiency greater than all baselines.

Figure A.5 – Normalized energy efficiency w.r.t AdaServ for 75% and 90% quality requirements (higher the better).

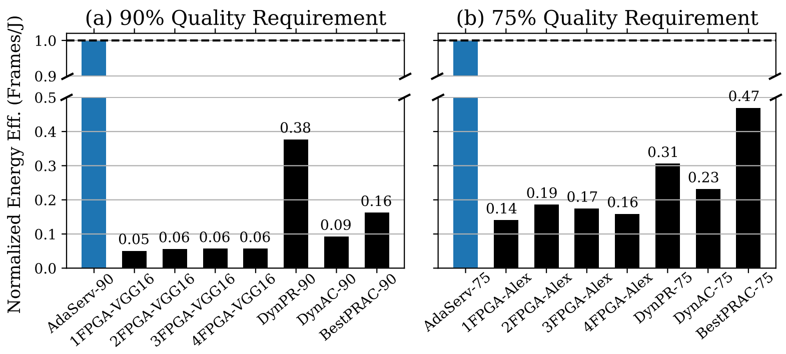


Fig. A.5 presents the energy efficiency (processed frames per Joule) of all baselines normalized w.r.t AdaServ (where AdaServ is one and high bars are better) for the 90% (a) and 75% (b) quantity requirements. AdaServ improves at least 5.38× the energy efficiency over state-of-the-art baselines (xFPGA-VGG16 and xFPGA-Alex) and at least 2.13× over the DynPR, DynAC, and BestPRAC baselines. For example, when compared to the best-performing not pruned VGG-16 on original CHaiDNN, 4FPGA-VGG16, AdaServ-90 processes 14.5× more inferences at 53% of the 4-VGG16 power dissipation with only 3.55% accuracy degradation across all inference requests. As for the 75% quality requirement, AdaServ-75 dissipates 65.57% and 34.26% of the 3FPGA-Alex and 4FPGA-Alex power while processing 4.75× and 3.63× more inferences, respectively, at the average cost of 4.83% in accuracy. Hence, we have shown that our two-step approach is capable of dynamically selecting configurations of higher performance and energy efficiency by allocating extra FPGAs and resource-hungry accelerators only when necessary.

APPENDIX B — EVALUATING THE USER-CONFIGURABLE ADASERV.

B.1 AdaServ with User-Configurable Runtime Search

To give the user more control over the throughput-power trade-off, we have implemented a second algorithm for selecting configurations in AdaServ (from Appendix A). It consists of a user-configurable search, which uses two parameters (predefined by the user) of an objective function for combining throughput (α) and power (β) as

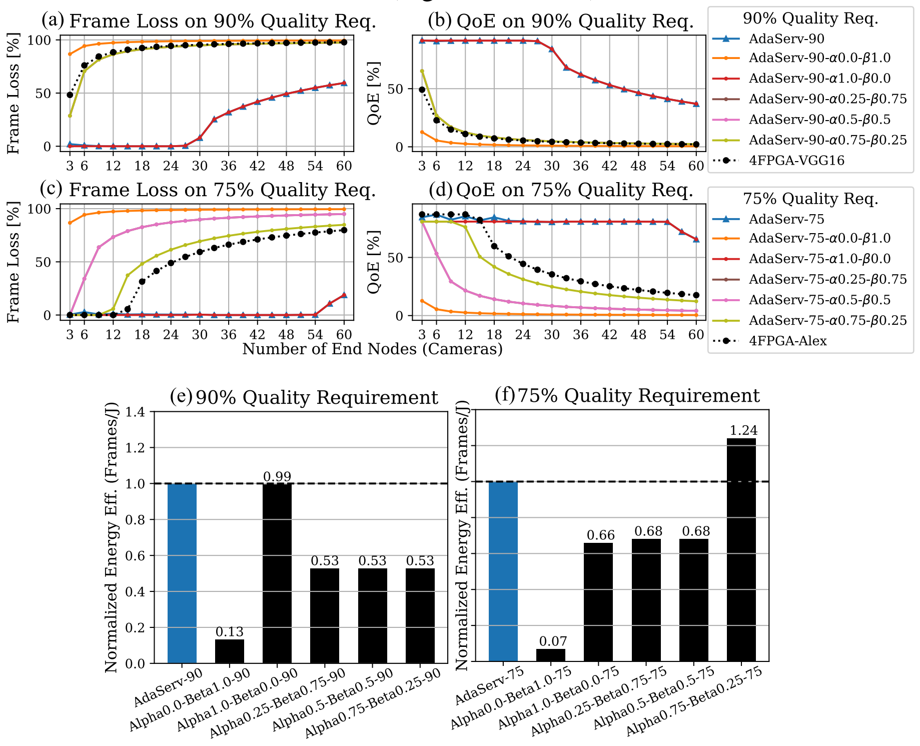
$$config^i_{value} = \alpha * config^i_{throughput} + \beta * (1 - config^i_{power})$$

where $config^i_{throughput}$ and $config^i_{power}$ are the i-th configuration's throughput and power normalized w.r.t all configurations in the library (min-max normalization). Then, the configuration that returns the best $config^i_{value}$ is selected (instead of the one giving the highest throughput, as done by Algorithm 3 in Appendix A). By tuning α and β values the user can, for example, set AdaServ to allocate all FPGAs, delivering full performance even though such throughput is not required by the current workload (e.g., α = 1 and β = 0). Alternatively, the user may also aim for power consumption (e.g. there is a certain power cap that must be respected), which will make AdaServ to select low power configurations, even if that means dropping frames (e.g., α = 0 and β = 1). We call AdaServ running with a user-defined objective function the "**User-Configurable AdaServ**."

B.2 Results

In this subsection, we show the benefits of giving the user control over the power-throughput trade-off in the User-Configurable AdaServ. The reader may also refer to the methodology presented in Section A.2 for further details on th evaluation setup. For that, besides the Automatic AdaServ, which was evaluated in previous sections, we also show the User-Configurable AdaServ on five different settings of α (the throughput weight) and β (the power weight). These settings range from a fully throughput-oriented selection of configurations ($\alpha = 1$ and $\beta = 0$) to a fully power-oriented ($\alpha = 0$ and $\beta = 1$) in 0.25 steps (i.e., $(1, 0)$, $(0.75, 0.25)$, $(0.5, 0.5)$, $(0.25, 0.75)$, $(0, 1)$). Fig. B.1 shows frame loss (plots (a) and (c)) and QoE (plots (b) and (d)) of these AdaServ settings on the same experiment from previous sections on both 75 and 90$\%$ Quality Requirements, respec-

Figure B.1 – Frame Loss (plots a and c) and QoE (plots b and d) for the Automatic and User-Configurable AdaServ with varied values of α and β for the 90% and 75% Quality Requirements, respectively. Best original baselines on four FPGAs are also shown for comparison. Normalized energy efficiency w.r.t AdaServ's Automatic Configuration in plots e and f (higher the better).



tively. User-Configurable AdaServ settings are indicated with α and β values, Automatic AdaServ is presented with no α and β values. The best performing original baselines (4FPGA-VGG16 and 4FPGA-Alex) are also shown for comparison.

Initially, let us discuss the User-Configurable AdaServ with $\alpha = 0$ and $\beta = 1$. With that user setting, AdaServ focuses solely on saving power, even if that means that frames are dropped. This setting makes AdaServ to allocate a single FPGA (with accelerator version 2 - see Table A.1) under either Quality Requirement. Consequently, both AdaServ-75-α0.0-β1.0 and AdaServ-90-α0.0-β1.0 (orange curves in Fig. B.1) end up showing the highest rates of frame loss. However, as was intended by the user, they also show the lowest levels of power dissipation: 81% and 82.4% less power than the Automatic AdaServ on the 75% and 90% Quality Requirements (blue curves in Fig. B.1, AdaServ-75 and AdaServ-90), respectively. Both AdaServ-75-α0.0-β1.0 and AdaServ-90-α0.0-β1.0 lose 98% of the frames (recall that AdaServ-75 and AdaServ-90 lose only 1.89% and 23.34% of inferences on average, respectively).

Conversely, the User-Configurable throughput-oriented AdaServ (red curves in Fig. B.1, AdaServ-75-α1.0-β0.0 and AdaServ-90-α1.0-β0.0) achieve the same levels of frame loss than the Automatic AdaServ. That is because, with α = 1 and β = 0, the Automatic AdaServ always chooses configurations that deliver the highest FPS, regardless of whether that configuration allocates more resources than would be actually necessary to attend the current workload, which could lead to a less energy-efficient solution but assuring that the server's full throughput is always kept. In contrast, the Automatic AdaServ achieves that same frame loss but allocating power-hungry accelerators and extra FPGAs only when required. Especially under the 75% Quality Requirement, that difference becomes greater: while the User-Configurable throughput-oriented AdaServ selects four FPGAs from the start, the Automatic AdaServ (Adaserv-75) allocates the fourth FPGA only for workloads with 18 end nodes or more. Overall, the Automatic AdaServ ends up processing the same number of inferences (within 1% difference) but dissipating 22.5% less power than the throughput-oriented AdaServ (AdaServ-75-α1.0-β0.0).

From Fig. B.1, we may also note that choosing α and β values allows the user to tune the throughput-power trade-off. For instance, we can see that when setting User-Configurable AdaServ to search for configurations considering that throughput is 3$\times$ more important than power (α = 0.75 and β = 0.25), it is possible to achieve a close-to-zero frame loss for up to 12 end nodes on the 75% Quality Requirement (gold curve, AdaServ-75-α0.75-β0.25, in Fig. B.1(c)) while dissipating 79.1% less power than the throughput-oriented setting and 73% less than the Automatic AdaServ (red and blue curves in Fig. B.1(c), respectively). However, from that point on, AdaServ-75-α0.75-β0.25 starts to lose frames since its configurations can no longer hold the increasing workload.

Still, the setting AdaServ-75-α0.75-β0.25 grants the highest energy efficiency over all evaluated configurable or automatic AdaServ settings (1.24$\times$ over the Automatic AdaServ-75): see Fig. B.1(e) and (f) for the normalized energy efficiency (frames processed per Joule). Plots follow the same idea of Fig. A.5 - all values are normalized w.r.t Automatic AdaServ (i.e., AdaServ-75 is the reference for settings under the 75% Quality Requirement and AdaServ-90 for settings under the 90% Quality Requirement). Therefore, in an Edge environment, it may be crucial to manually set the server throughput-power trade-off. For example, the user may sacrifice some of the server performance to a full power-oriented setting like α = 0 and β = 1 that achieves minimal power dissipation; or a more balanced setting such as α = 0.75 and β = 0.25 that results in a highly energy-efficient setting.

APPENDIX C — EVALUATING HIGHER RESOLUTION IMAGES

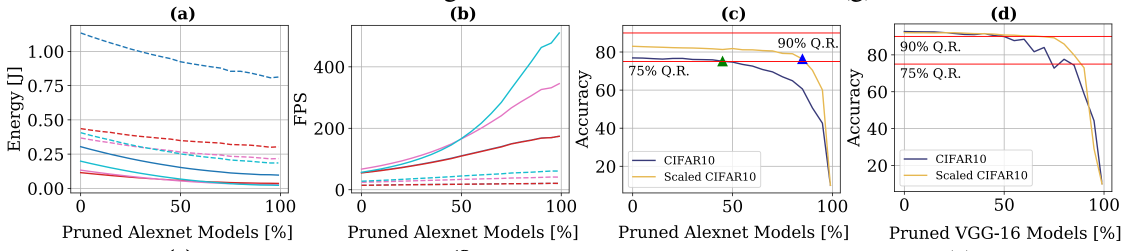
We have also experimented with images of higher resolution. This experiments were carried out within the HLS and pruning from AdaServ (Appendix A). This appendix presents such results.

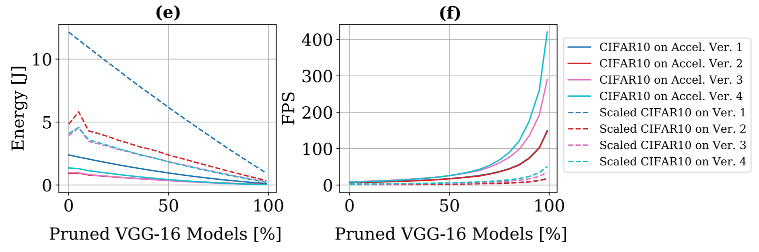
First, we note that our approach can work with images of any size since its modularity enables pruned CNN models trained for any particular input size to be dynamically loaded for inference (as any of the pruned CNN models already in use). However, we have opted for letting datasets with larger images out of previous evaluations because their elevated inference latency dominates the results (over the remaining datasets with 32x32 images) on frame loss and QoE on the evaluated scenarios. The reader may also refer to the methodology presented in Section A.2.

Therefore, we dedicate this final subsection to evaluate AdaServ on serving inferences on images of higher resolution. To that end, we have trained and pruned the original VGG-16 and AlexNet on the dataset that showed the lowest accuracy in our experiments, CIFAR-10, with images scaled from 32x32 (CIFAR-10's image format) to 96x96. We call this new dataset Scaled CIFAR-10. Training and pruning on the Scaled CIFAR-10 followed the same parameterization from other experiments, and images were resized to 96x96 using the bilinear interpolation method available in PyTorch. Fig. C.1 presents how the design space enlarges for CIFAR-10 when considering two image sizes. Energy and throughput (FPS) are given for pruned AlexNet models (plots (a) and (b)) and pruned VGG-16 models (plots (e) and (f)). Fig. C.1(c) and (d) compare the TOP-1 test accuracy of pruned AlexNet and VGG-16, respectively, on original CIFAR-10 and Scaled CIFAR-10 datasets. Fig. C.1(g) presents the averaged frame loss and QoE (75 ad 90% Quality Requirements) of Automatic and User-Configurable throughput-oriented AdaServ settings over the same experiment from previous sections but with all nodes requesting Scaled CIFAR-10 inferences only (best performing original baselines are also shown).

In our evaluations with Scaled CIFAR-10, both Automatic and Configurable throughput-oriented AdaServ (the best performing systems) lose over 90% of the inferences for 18 end nodes or more. Those relatively high rates of frame loss also cause low QoE results for all evaluated systems, including the best performing ones (Automatic and User-Configurable throughput-oriented AdaServ), as can be seen in Fig. C.1(g). We point out that such heavy workload can only be met by (i) adding more FPGAs to the server or (ii)

Figure C.1 – Energy and FPS for the four accelerator versions running AlexNet (plots a and b) and VGG-16 (plots e and f) models on CIFAR-10 and Scaled CIFAR-10, with accuracy in plots (c) and (d). Average Frame Loss and QoE over a full run with Scaled CIFAR-10 of Automatic and User-Configurable AdaServ and baselines in (g).





(g)

	Avg. Frame Loss	Avg. QoE
AdaServ-90	89.91%	9.1%
AdaServ-90-α1.0-β0.0	89.91%	9.1%
4FPGA-VGG16	98.55%	19.64%
AdaServ-75	47.96%	40.02%
AdaServ-75-α1.0-β0.0	47.77%	39.87%
4FPGA-Alex	76.34%	19.64%

reducing the quality requirement. AdaServ could accomplish both alternatives. The former is possible thanks to the addition of extra FPGA boards being transparent in AdaServ (since inference requests are sent over the network to be processed at specific boards). The latter is possible by setting a lower quality requirement, which would allow selecting pruned CNN models of lower accuracy that run faster, returning higher throughput (a lower quality requirement could be run in the same way that 90% and 75% quality requirements are executed).

To have a better understanding of these results with a larger image resolution, let us look at the design space created by including up-scaled images: when comparing the original CIFAR-10 to our Scaled CIFAR-10, we see reductions in the accelerators' throughput that range from 3 to 23.3× (considering execution of pruned VGG-16 models) and from 2 to 8.4× (considering pruned AlexNet models), see Figures C.1(b) and (f). Energy consumption follows the same behavior in Figures C.1(a) and (e). This cost in throughput (and energy) is mainly caused by the larger time taken by transferring larger images; and the larger feature maps that need to be transferred among FPGA and main memory between the execution of consecutive convolutional layers. Since CNN layer parameters are kept the same (i.e., same filter size and stride), these convolutional layers will produce output feature maps that are "taller" and "wider" (while keeping the same number of channels) when processing larger input images. However, the increased time to process each inference comes with significant improvements in accuracy.

Gains in accuracy are especially noticeable among AlexNet models. The original

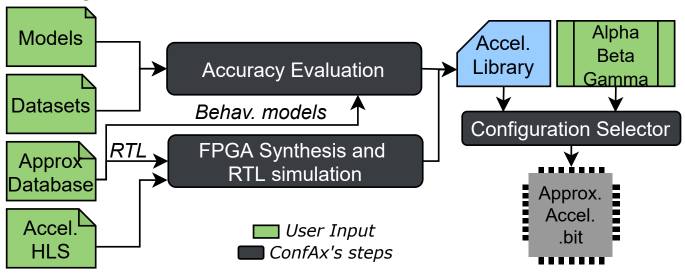
AlexNet model improved its accuracy from 76.9% to 83.01% with 96x96 images. For the original VGG-16, on the other hand, scaled images did not improve accuracy: the original VGG-16 showed accuracy 0.61% below the 92.65% achieved with 32x32 images. Nevertheless, we note that this accuracy could be improved by specially tuning the training process for VGG-16. Besides the accuracy of original CNN models, using larger images has the interesting effect of keeping the accuracy curve flat for more aggressive pruning rates on both CNN models (i.e., image up-scaling also increases the accuracy of pruned CNN models). For example, AlexNet on 32x32 images stays above the 75% Quality Requirement (75% Q.R. in Fig. C.1(c)) from 0% to 45% (green triangle in Fig. C.1(c)). On the other hand, when processing 96x96 images, AlexNet stays above the same Quality Requirement until 85% of its channels are pruned out (blue triangle in Fig. C.1(c)). Similar behavior is observed for VGG-16 in Fig. C.1(d). That means that up-scaling the input image makes more pruned CNN models available for inference given a certain accuracy threshold, enlarging AdaServ's design space.

APPENDIX D — A STUDY ON APPROXIMATE COMPUTING FOR FPGA-BASED ACCELERATORS

For this thesis, we have also explored approximate computing (i.e., approximate multipliers - as in (CHAKRADHAR; RAGHUNATHAN, 2010)) as an optimization axis for delivering efficient CNN inference processing on FPGAs. *However, we have opted to not carry out this optimization further in the thesis. The use of approximate multipliers has shown restricted advantages when applied on FPGA accelerators and precluding an "off-the-shelf" use of the SoTA quantization techniques (e.g., Brevitas Quantization-Aware Training and the FINN accelerators).* In this appendix we report our results on what-would-be the Approximate Computing workfront.

D.1 ConfAx

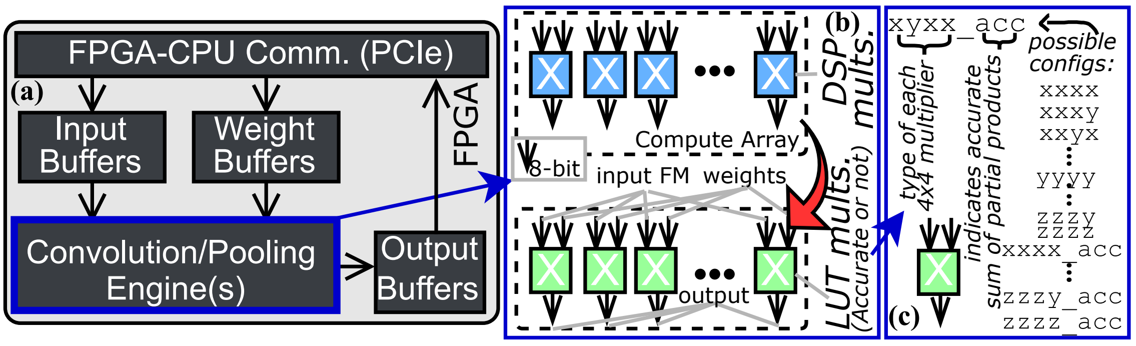
Figure D.1 – Workflow for the ConfAx workfront.



This exploration on AC was motivated by state-of-the-art works (AYHAN; ALTUN, 2019; YOUSSEF et al., 2020; CASTRO-GODÍNEZ et al., 2020; WANG et al., 2019) showing that, when enabled by FPGAs, AC can be used to optimize CNN accelerators to single optimization goals (e.g., low-power). However, the IoT-Edge presents varied application environments, requiring **multi-target optimizations** (e.g., scenarios of high workload, demanding maximum performance or scenarios constrained by energy). In this context, we undertook the use of AC for multi-target optimization of FPGA-based approximated CNN accelerators. We called it ConfAx for Configurable use of Approximate multipliers for FPGA-based CNN acceleration. The general idea lies in (smartly) replace the original multipliers in a single-engine accelerator (CHaiDNN-V2 (Xilinx Inc, 2020)) per approximate ones.

ConfAx was implemented covering only a design-time phase and, thus, contains a Configuration Selector to explore the design space (see Figure D.1 for ConfAx's work-

Figure D.2 – (a) Schematic of the modified CHaiDNN-v2; (b) DSPs replaced by LUT multipliers; (c) and, an example of selected approximate configuration.



flow). To build the library in this workfront, ConfAx performs two steps: *Accuracy Evaluation* and *FPGA Synthesis and RTL Simulation*. Besides the CNN descriptions with their training datasets, original accelerator code - the single-engine CHaiDNN-v2, ConfAx receives the approximate multipliers database (containing RTL implementations and behavioral models - Approx Database in Figure D.1). As ConfAx only replaces accurate by approximate multipliers in the original accelerator design (see D.2(a)), it starts by evaluating their impact on the CNN inference accuracy (Accuracy Evaluation). To that end, ConfAx uses a dedicated software implementation for the convolutional layer that considers a multiplier's behavioral model to insert its error into the convolution output. Then, ConfAx runs the CNN inference on all test images for every approximate multiplier to gather the CNN accuracy produced by each multiplier. Inference accuracy values will be kept to support the following steps in the workflow.

In the next step (FPGA Synthesis and RTL simulation), ConfAx takes the original user's accelerator *as a template* to replace the original/accurate multipliers with approximate multipliers (further explained). Resulting in multiple accelerator versions, each one is configured with a specific approximate multiplier (in our case study, we are synthesizing all convolution-related multiplications in the original accelerator's code with the same approximate multiplier). Later, ConfAx synthesizes and performs RTL simulations to gather power and performance results for every Accelerator Version. After all versions have been synthesized and their accuracy evaluated, they are stored in the library (Figure 4.1). The Configuration Selector uses the library that selects the approximate accelerator version that best matches the user-defined optimization targets (further explained). Finally, the Configuration Selector will output a single accelerator version that can be deployed to the FPGA for serving inferences at the Edge. Next, we detail the main components of the ConfAx workflow.

D.1.1 Approximate Accelerators

We have adopted the open-source SMApproxLib (ULLAH; MURTHY; KUMAR, 2018) for configurable and automatic generation of approximate multipliers optimized for FPGA-based designs. By replacing the original DSP-implemented multiplications (DSP mults. in Figure D.2(b)) with the RTL code of LUT-based approximate multipliers from SMApproxLib (LUT mults. in Figure D.2(b)), we enable a library of approximate CNN accelerators with varied accuracy, power, and performance profiles. It is important to note that all accelerator circuitry surrounding the multipliers remains unchanged. Naive sign converters were added to multipliers' inputs and output so that the unsigned multiplication could be carried out and the output converted afterward if necessary.

SMApproxLib leverages the FPGA 6-input LUTs and fast carry chains for designing power-efficient and fast multipliers. Precisely, SMApproxLib builds a $N \times N$ multiplier by recursively instantiating $N/2$ multipliers. Therefore, for the 8-bit multiplications used in our case study, four 4×4 instances will be in each multiplier. Each of those instances can be configured with either accurate (design `A`) or one of the three SMApproxLib approximate designs:

- Design `x` (Approx1 in (ULLAH; MURTHY; KUMAR, 2018)) that uses approximate additions for reducing the partial products;
- Design `y` (Approx2 in (ULLAH; MURTHY; KUMAR, 2018)), which is optimized for latency and energy, that removes carry chains used in the accumulation of partial products;
- Design `z` (Approx3 in (ULLAH; MURTHY; KUMAR, 2018)), similar to `y`, predicts the carry out from preceding bit locations to improve accuracy.

These approximate designs implement different LUT configurations presenting different critical path delays, power, and error. SMApproxLib also allows for accurate or approximate accumulation of the $N/2$ partial products. The **naming system** used to identify the SMApproxLib's approximate configurations is $\mathtt{m_1m_2m_3m_4\_acc}$, where $\mathtt{m_i}$ indicates the design of the i-th 4×4 multiplier (`x`, `y`, `z`, or `A`) and `_acc` at the end of the configuration name indicates the accurate sum of the $N/2$ partial products - see Figure D.2(c) also. In summary, there are $4^4 \times 2 = 512$ possible 8×8 multipliers, and each is used to generate one accelerator version.

Table D.1 – Accelerator versions resource usage (XCU200 FPGA).

	BRAM	DSP	FF	LUT
CHaiDNN-v2	26.7%	17.65%	5.03%	8.91%
LUT Accurate (`AAAA_acc`)	26.7%	2.7%	6.67%	18%
ConfAx (`yzzz_acc`)	26.7%	2.7%	6.42%	14.6%

D.1.2 Configuration Selector in ConfAx

To navigate the design space created by the accelerator library, ConfAx employs a multi-target profit equation to sort and select accelerators. With this configurable profit equation, the user can control the performance-power-accuracy trade-off and tailor the design to any optimization goal or environmental condition. This is achieved employing three configurable parameters to combine throughput (α), power (β), and accuracy (γ) as

$$profit_i = \alpha * config^i_{throughput} + \beta * (1 - config^i_{power}) + \gamma * config^i_{accuracy} \tag{D.1}$$

where $config^i_{throughput}$, $config^i_{power}$, and $config^i_{accuracy}$ are the i-th accelerator version's throughput (in Frames Per Second - FPS), power dissipation, and CNN's accuracy normalized w.r.t all versions in the library (min-max normalization). See (α,β,γ) in Figure D.1 Configuration Selector. After the profit of all configurations has been calculated and sorted, the Configuration Selector outputs the bitstream file of the accelerator version with the highest profit, providing the best match to the user's requirements. After that, the accelerator is deployed, and inference processing can start.

D.2 Results

D.2.1 Evaluation Scenario

For this workfront, we use the single-engine CHaiDNN-V2. Accelerator versions that do not achieve 300Mhz (Section 4.1) are synthesized at 222MHz (which is the maximum frequency achievable by the accurate LUT-based multiplier, configuration `AAAA_acc`). Out of the 512 configurations evaluated, 99 achieved 300Mhz, 410 achieved 222Mhz, and three were not included in ConfAx library since they violated the 222Mhz threshold. Table D.1 reports the FPGA resource usage for the CHaiDNN-v2 baseline, accurate LUT-based version, and a representative approximate version. To evaluate the

Table D.2 – Results for the Edge case study in (a) and ConfAx gains over CHaiDNN in (b) for all selected versions on each evaluated tuple.

			(a) Edge Case Study			(b) gains over CHaiDNN			
(α,β,γ)	Tuple Description	Selected Version	Frame Loss	CNN Accuracy	QoE	Power	Energy	EDP	Density
(1,0,0)	Full FPS	`zxzx` (300MHz)	**3.25%**	74.30%	71.89%	1.366×	1.366×	1.366×	1.003×
(0,1,0)	Full Power	`zyzy_acc` (222MHz)	15.20%	74.30%	63.01%	**1.646**×	**1.443**×	1.265×	0.956×
(0,0,1)	Full Accuracy	`xzAz` (222MHz)	15.20%	**74.70%**	63.35%	1.642×	1.440×	1.262×	0.893×
(0,1,1)	Power + Acc.	`yzxz_acc` (222MHz)	15.20%	**74.70%**	63.35%	1.644×	1.441×	1.263×	0.916×
(1,0,1)	FPS + Acc.	`zxxx` (300MHz)	**3.25%**	74.50%	72.08%	1.368×	1.368×	1.368×	1.020×
(1,1,0)	FPS + Power	`yzzz_acc` (300MHz)	**3.25%**	74.50%	72.08%	1.372×	1.372×	**1.372**×	**1.085**×
(1,1,1)	All Important	`yzzz_acc` (300MHz)	**3.25%**	74.50%	72.08%	1.372×	1.372×	**1.372**×	**1.085**×
`AAAA_acc` (222MHz, LUT accurate)			15.20%	**74.70%**	63.35%	1.630×	1.428×	1.252×	0.773×
CHaiDNN (BASELINE)			**3.25%**	**74.70%**	**72.27%**	1×	1×	1×	1×

error due to approximate units, we modified PyTorch's convolutional layer with a custom implementation that mimics the approximate hardware: after the standard im2col is applied to the input, it is multiplied with the weight matrix using the multiplier's behavioral model from SMApproxLib for each element-wise multiplication so that the error can be correctly propagated through the whole forward pass.

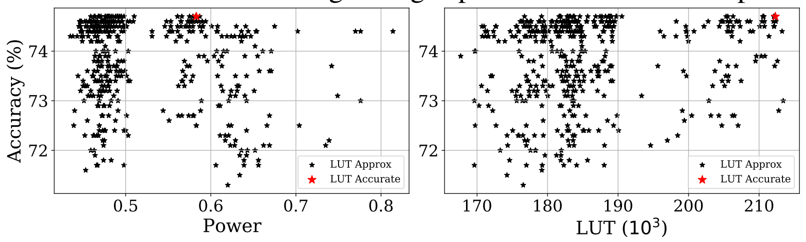
Our case study is based on a typical smart video surveillance system (ZHANG et al., 2015b) that deals with numerous cameras (IoT devices) continuously requesting inferences (frames) to a local Edge server, where the inference processing is offloaded to the FPGA. Therefore, we have set 60 IoT devices to request inferences at the real-time rate of 30 FPS. All evaluations are 10 seconds long. ConfAx selects accelerators based on user-defined α, β, and γ parameters. We note that those parameters can be tuned to specific applications and user goals. Here, we take seven representative settings to show ConfAx potential (presented in Table D.2 with their description). Next, we evaluate ConfAx over the original CHaiDNN-v2 accelerator (baseline) on performance, power, energy, and Quality of Experience (QoE - accuracy times the percentage of processed frames) (CARTAS et al., 2019; HUYNH-THU; GHANBARI, 2008). With QoE, we assess the user experience that targets low frame loss levels and high accuracy levels. We also present results on Performance Density (FPS per LUT) to assess how area-efficient an FPGA design is[1].

D.2.2 Evaluation

Table D.2 evaluates ConfAx under different optimization tuples (α, β, γ). The 'Tuple Description' column describes their goal. The 'Selected Version' column gives the accelerator version selected by ConfAx among the library's 512 versions for each of these tuples (accelerator versions are identified with the **naming system** presented

[1]For Density, we assume that a DSP slice takes the area of 10 LUTs.

Figure D.3 – Power and LUT usage design space of LUT-based compute arrays.



earlier. Table D.2 also reports frame loss, CNN classification accuracy, and QoE over the 10 seconds run for the Edge smart surveillance case study ((a) columns) as well as the (b) columns presenting improvements in power, energy, Energy-Delay Product (EDP), and Performance Density over the baseline (the accurate CHaiDNN-v2 with multiplications executed on DSP slices). The best values for each evaluated metric are highlighted.

We see that the CHaiDNN-v2 baseline achieves the highest QoE since it grants the lowest fame loss (no LUT-based multiplier could achieve throughput above what is delivered by DSP slices) with the CNN's original accuracy. However, as shown in Table D.2, the full CNN accuracy is also achieved by LUT-based multipliers (e.g., versions `AAAA_acc` and `xzAz`), while returning 1.63× and 1.642× reductions in power dissipation, respectively. Contrarily, with approximate versions such as `zxzx`, `zxxx`, and `yzzz_acc`, it is possible to match the baseline's throughput with major power reductions (up to 1.646×). Overall, ConfAx produces approximate accelerators with power dissipation and energy consumption lower than the baseline for all evaluated tuples at small accuracy drops.

ConfAx's optimization tuple enables full control over the hardware generated. For instance, for tuples with high α values (the performance parameter), ConfAx will deploy accelerators of maximum throughput. On the other hand, in a power-constrained scenario, the user may set β = 1 to deploy a low-power accelerator: version `zyzy_acc` that ends up saving 1.646× the power dissipated by the baseline. We also note that combined optimization targets are possible. For example, tuple (1,1,0) sets ConfAx to search for energy-efficient accelerators since it focuses on lowering the power dissipation with maximum performance (at the cost of accuracy). The deployed version for this tuple, `yzzz_acc`, ends up returning the best EDP (as well presenting the highest density) among all evaluated systems. To give a better understanding of the differences between the accurate and approximate LUT-based multipliers in ConfAx's library, Fig. D.3 details the power and LUT usage design space of the compute arrays used in this work (see Fig. D.2). Design points represent compute arrays with all multipliers (1024) used inside CHaiDNN-v2's convolution engine (see Fig. D.2(b)). From Fig. D.3, we can see that the ConfAx library with its approximate versions enables new pareto points that, due to their varied accuracy and power/resource profiles, can be used for configuring the inference processing at the Edge, where those multiple optimization targets are required to cover all possible application environments.

We also note that the approximate accelerators cause relatively low accuracy losses (0.88% on average) for the CNN's classification task compared to the baseline (74.70% accuracy - see Table D.2). In conclusion, ConfAx not only enables full control over the hardware generation through the use of an efficient Configuration Selector, but also produces energy and power savings on all evaluated approximate accelerators.

APPENDIX E — A STUDY ON THE DESIGN SPACE OF SPLIT INFERENCE

Before developing the offloading mechanism presented in the last workfront (Section 4.5), a design space combining quantization, pruning, and split inference was carried out. As done in some works of this thesis, the navigation of this design space is performed through a user-configurable profit equation. The steps from the construction to the exploration of the design points constitute a framework. Below, we present it as well as its evaluation on an IoT application scenario.

E.1 Framework

Figure E.1 – Workflow for constructing and searching the design space.

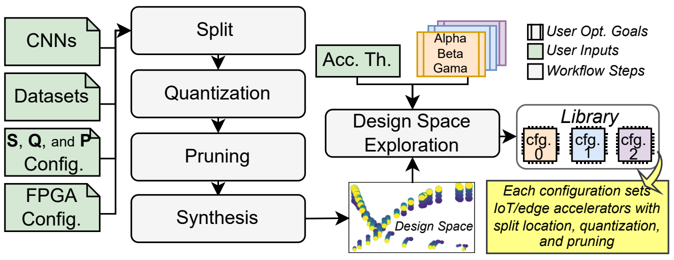


Figure E.1 shows the workflow for creating and navigating the design-space. It follows the general steps taken in the main workfronts of this thesis. S, Q, and P represent configuration files for setting the split (S), quantization (Q), pruning (P) for training and synthesizing the accelerators. The framework delivers a set of design *configurations* (defining the level of each optimization), each tailored for one user-informed optimization tuple (Alpha, Beta, and Gamma). As will be explained later, these optimization tuples are the weights of a profit equation that combines latency, accuracy, and power for sorting the design versions.

E.1.1 Split

In the proposed framework, the user is responsible for defining the range S of possible split locations (i.e., after which layers offloading is possible). The split step will take these configuration and generate a *split CNN* for each possible location. For example, defining $S = \{1, 2, 3, ..., L\}$ for a CNN of L layers, means that after splitting the inference after every layer is possible. Here, we are defining the *IoT layers* as the initial layers that run locally on the IoT device. After the last IoT layer (split location), the output data (feature map) is sent over the network for the edge server running the set of *edge layers*.

A split CNN is created by instantiating a new Brevitas/PyTorch CNN model. This new split CNN will import the same parameterization of the user-supplied CNN but with a small modification. First, IoT and edge layers are separated into two independent sets of layers (i.e., two `ModuleList()` in PyTorch). This separation will facilitate training, having different quantization and pruning levels, and will also allow these layers to be synthesized as two FPGA accelerators. The Split Step will iterate over all possible split locations to create a pair of IoT/edge layers for each split location, feeding the next step in the framework.

E.1.2 Quantization, Pruning, and FPGA Synthesis

The design flow for quantizing, pruning, and synthesizing the CNNs uses the work developed for the thesis. In this study, however, we treat these optimizations as parameters (as done for the split locations S).

For the Quantization Step, the framework takes in the user-defined set of possible quantization values Q. In this framework, only the IoT layers are optimized with the more constrained quantization levels, leaving the edge layers (that will be deployed at the edge server) with the largest bit-width in Q. The reason for this is two-fold: first, we assume, in this work, that splitting the inference from IoT to edge implies in sharing the computation from a more restricted to a less restricted device; and, second, since training the CNNs is a timely process, limiting the number of combinations to train was necessary. Let us take $Q = \{1, 2, 4\}$ as an example. For this, three versions of each IoT set of layers will be trained with edge layers at 4-bit quantization.

Following the reasoning in the quantization step, only the IoT layers are pruned. For this step, the user must also inform the set P of pruning rates that the IoT layers must be pruned with. For example, $P = \{0, 25, 50, 75\}$ will return a not-pruned version (0% pruning rate) and three other pruned versions, from 25 to 75% pruning rates, for every quantized CNN (e.g., 0% pruning for 1, 2, and 4 bits, 25% for 1, 2, and 4, and so on).

For the FPGA synthesis, IoT and edge layers are synthesized following the user configuration for each node (e.g, a smaller Pynq-Z1 board for IoT and a larger ZCU104 to be used at the edge server). For splitting FINN accelerators, a small modification to the original workflow was required. It takes the split location to set new interfaces (as AXI stream ports, default in FINN): a new output for the IoT layers (making it a new dataflow from the original input up to the split location), and new input for the edge part (that goes up to the last CNN layer). It results in two separate accelerators for synthesis. This step, besides delivering FPGA bitstreams for IoT and edge, performs RTL simulations to asses IoT and edge performance.

E.1.3 Design Space Exploration

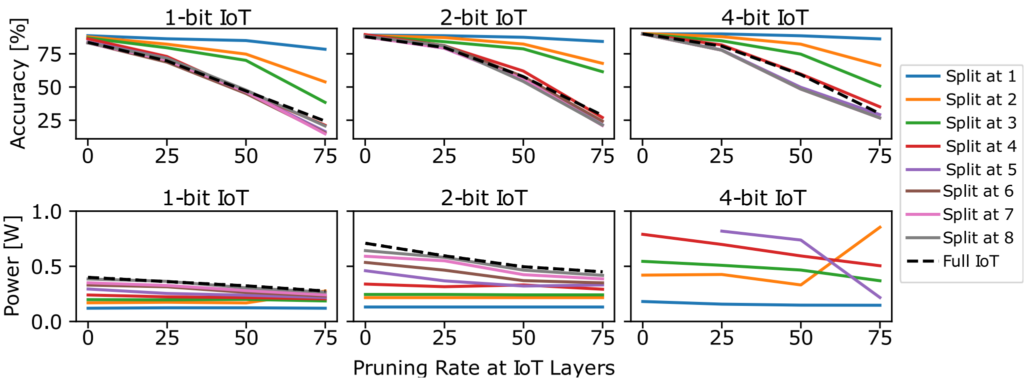
The Design Space Exploration completes the framework. It systematically explores the design space to find the design version that best fits the user requirements (expressed as the optimization tuple). This search is guided by a profit equation, combining accuracy (α), latency (β), and power (γ) on the three user-specified weights, as follows:

$$\begin{gathered}\text{profit}_i = \alpha \cdot \text{config}^i_{\text{accuracy}} + \\ \beta \cdot (1 - \text{config}^i_{\text{latency}}) + \gamma \cdot (1 - \text{config}^i_{\text{power}})\end{gathered} \tag{E.1}$$

where $\text{config}^i_{\text{latency}}$, $\text{config}^i_{\text{power}}$, and $\text{config}^i_{\text{accuracy}}$ are the i-th configuration's latency, power dissipation, and accuracy normalized w.r.t all configurations in the design space (min-max normalization).

The search begins by excluding all design configurations with accuracy below the user-specified minimum (*Acc. Th.* in Figure E.1). Then, profit equation is evaluated for the remaining configurations. These points are sorted, and the configuration with the highest profit is delivered. If the user provides multiple tuples, a library of optimal design configurations is generated, with one configuration per tuple. These configurations represent the *pareto front* for the specified optimization goal and constitute the main output of this framework. It is the user's responsibility to leverage this library effectively. For instance, for the case of bad networking (which would increase the IoT-edge communication overhead), the user can set a tuple with a high β value - resulting in a low-latency inference processing to compensate for elevated communication delays. On the other hand, when networking is at a good state, more weight can be given to accuracy, for instance, increasing the overall user experience.

Figure E.2 – Accuracy (upper plots) and Power (bottom) for split CNV with IoT layers at 1, 2, and 4 bits quantization w.r.t pruning rates from 0 to 75% (edge layers run not-pruned 4-bit). Curves represent split locations from layer 1 to 8. not shown 4-bit configurations do not fit the IoT FPGA.



E.2 Methodology

Our use-case application is an IoT smart video application with an IoT device connected over cellular network to an edge server. Evaluations run over the traces from the (KOUSIAS et al., 2023) 5/6G dataset. Results use the best (top 1% percentile of the recorded latency/throughput values), average, and worst (bottom 1% on latency/throughput) networking conditions of the dataset.

We utilized the CNV CNN from FINN on the CIFAR-10 dataset with quantization from 1 to 4 bits (4-bit quantization was the largest CNV fitting the FPGA) in our experiments. For the layers running on the IoT device, four pruning rates were evaluated (0, 25, 50, and 75%). Accuracy is reported on Brevitas TOP-1 test accuracy.

Two baselines are used: **Full IoT** for the original CNV (from 1 to 4 bits, quantization indicated) running fully locally on the IoT FPGA that represents an IoT-only SoTA solution; and **Full Edge** that fully offloads all inferences (i.e. input images are sent) to the edge running an original CNV at 4-bit quantization, representing a SoTA edge processing. For all evaluations, an accuracy threshold (Min Acc in Fig. E.1) of 15% below the original 4-bit CNV accuracy was set as well as six $\{\alpha, \beta, \gamma\}$ tuples (presented below).

E.3 Results

Figure E.2 presents the accuracy (top plots) and power (bottom plots) for the CNV models at 1, 2, and 4-bit quantization (each has a plot) w.r.t pruning rates ranging from 0 to 75% for all possible splits in the CNV (layers 1 to 8). The number of the split indicates

Table E.1 – Optimization tuples with the configuration Quantization, Pruning, and Split parameters.

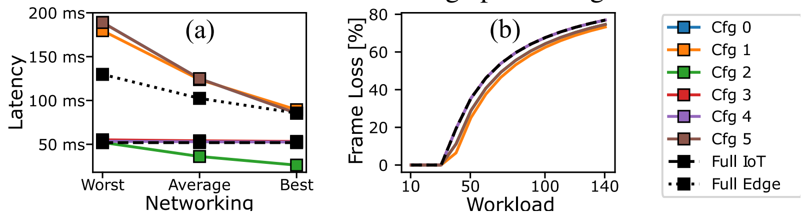
Cfg.	Opt. Tuple	Q	P	S	Cfg.	Opt. Tuple	Q	P	S
0	{0.5,0.5,0.0}	4	75	1	4	{0.0,0.0,1.0}	1	0	8
1	{1.0,0.0,0.0}	4	0	2	5	{0.3,0.3,0.3}	4	75	1
2	{0.0,1.0,0.0}	1	75	1	Full IoT	-	1	0	-
3	{0.5,0.0,0.5}	2	0	8	Full Edge	-	4	0	-

after which layer the inference continues onto the edge server. The inference executed completely on the IoT device is represented in the Full IoT curve. Recall from earlier that only the layers running on the IoT device get pruned and the layers on the edge server are always at the larger bit-width (4-bit quantization). Regarding accuracy, we can notice that splitting at the foremost layers returns the highest accuracy levels: splits 1, 2, and 3 (blue, orange, and green curves, respectively) - never below 86.95% when not pruned. For the sake of comparison, running all layers on the edge under no pruning and 4-bit quantization returns 90.04% accuracy. Given that only layers before the split get pruned, for those early splits, the layers running at the edge server can hold the accuracy at higher levels. For instance, even under the most aggressive compression (1-bit quantization, 75% pruning), splitting after the first layer (split 1) returns accuracy of 78.43%, much higher than the 24.36% delivered by the original CNV (dashed curve) on Full IoT when pruned at 75%. On the other hand, for later splits (after layer 4 onwards), the layers at the edge server are not able to hold accuracy. It becomes clear that there is a threshold on the split position in which pruning/quantization can be more aggressively applied at no major costs in accuracy.

The bottom plots in Figure E.2, for the sake of simplicity and given that the optimizations are being applied on the IoT side only, present the power dissipated at the IoT board. Naturally, all split points present power levels below the Full IoT, which executes all layers locally. The early is the split location, the less power is dissipated at the IoT device. It is also worth mentioned that not all designs can be deployed at the IoT board since it is a small device, aimed for embedded applications. For the 4-bit quantization, only CNVs split up to layer 5 fit the IoT holding a Pynq-Z1 FPGA. For a comparison, the FPGA on the edge server, dissipates from 0.610W (split 8) to 1.599W (split 1). For the edge server, the opposite reasoning holds, as earlier the split location is, the more layers get offloaded to the edge, increasing its power dissipation.

The accuracy and power overview presented earlier suggests us that defining a trade-off among these metrics (and performance as seen next) may help to navigate the

Figure E.3 – (a) Latency under three networking corner cases. (b) Frame loss (lower, the better) for varied workloads. Plots consider inferences with configurations (Cfg.) from 0 to 5 as well as Full IoT and Full Edge processing.



many possible configurations for inference. In our work, the user can express the trade-off by setting the $\{\alpha, \beta, \gamma\}$ optimization tuple, weighting accuracy, latency, and power, respectively, for searching design configurations. For illustration purposes, we present six optimization tuples ranging from full latency-oriented (setting β to 1 and remaining to 0) to full accuracy, power, and also other mixed optimization tuples (see Table E.1). Labels from 0 to 5 are used to identify the tuples and their generated configurations.

Figure E.3(a) gives the latency of the configurations generated and the two baselines for the three corner cases on the IoT-edge communication. You may note the impact of the communication delay mainly for configurations 1 and 5 (orange and brown curves) and for the Full Edge baseline (that must offload the whole input image before processing it). The lowest latency, however, is delivered by configuration 2 (Cfg. 2, green curve), which was generated by the $\{0.0, 1.0, 0.0\}$ tuple (fully latency-oriented). This tuple runs a single layer (split 1) at 1-bit quantization and 75% pruning rate on the IoT. Figure E.3(b) presents the Frame Loss, the rate of lost frames, for workloads ranging from 10 to 140 Frames Per Second (FPS, x-axis) for all configurations and baselines. In this case, we may note that the configuration delivering the best throughput is not configuration 2 (as in the latency plot) but configuration 1 running locally 4-bit not-pruned layers (split 2). The reason for this higher throughput is due the edge delivering a higher FPS in configuration 1. With this brief overview, it becomes clear that when considering the highly volatile IoT-edge environment and multi-objective user requirements, there is no single configuration for an optimized inference processing. Thus, careful navigation of the design space must be taken to cover the required user needs.

APPENDIX F — MODIFICATIONS TO THE FINN CODEBASE

Here, we provide a walkthrough on our modifications to FINN to implement pruning, early-exiting, and splitting. *Refer to FINN* ***under version v0.5b*** *in https://github.com/-Xilinx/finn for the code mentioned here.*

F.1 For the pruned CNNs

In this work there are two possible implementations for pruned CNNs on FINN. One is the the traditional FINN design-flow starting from an ONNX with a pruned CNN. This is the so-called Fixed-Pruning Accelerators in Section 4.2. They cannot change the pruning the rate at runtime and follow the original design-flow. In Section Section 4.2 we also presented the Flexible-Pruning Accelerators that can change the pruning rate at runtime. The modifications here concern this last type of accelerator.

F.1.1 Setting up

FINN uses templated C++ classes for implementing the HLS modules. For example, FINN's main class that implements matrix multiplications for convolutional and fully-connected layers, the MVTU (`Matrix_Vector_Activate_Batch`) has the following header:

```
template<
  unsigned MatrixW, unsigned MatrixH, unsigned SIMD, unsigned PE, unsigned MMV,
  typename TSrcI = Identity, typename TDstI = Identity, typename TWeightI = Identity,
  typename TI, typename TO, typename TW, typename TA, typename R
>
void Matrix_Vector_Activate_Batch(hls::stream<TI> &in,
  hls::stream<TO> &out,
  TW  const &weights,
  TA  const &activation,
  int const  reps,
  R const &r) { ... }
```

Despite configuring the parallelization with `SIMD` and `PE` (and some auxiliary type parameters), the class receives the `MatrixW` and `MatrixH`. These two parameters configure the folding of the MVTU. They are calculated from the input and output feature maps and are used in the MVTU to define the loop bounds and memory accesses. Keep in mind that, as seen in the background chapeter, the input of MVTU has been already adaptaed by the Sliding Window Unit (SWU) so the convolution can be performed as a matrix multiplication. So, `MatrixW` equals the width of the input matrix, which means the number of output channels; and, `MatrixH` equals the heigth of the input matrix, which means the number of input channels $\times$ k^2, where k is the filter width/height.

To make the accelerator able to change the pruning rate, we repurposed `MatrixW` and `MatrixH` in the parameters list and added signals to the MVTU's interface:

```
..., volatile unsigned &MatrixW_current,
volatile unsigned &MatrixH_current, ...
```

Now, the original parameters give the worse-case loop bounds and the signals (ending on `_current`) give the current model size, making it possible to exit the loop before reaching its original bound. That enabled to control the loops from 'outside'. The same was done for all HLS classes that had any parameter controlled/affected by the number of input/output channels. We added a `_flexible` to those classes (namely, `Matrix_Vector_Activate_Batch_Flexible`, `ConvolutionInputGenerator_Flexible`, `StreamingDataWidthConverter_Batch_Flexible`, `StreamingMaxPool_Precision_Flexible`). All new HLS classes were verified and passed the tests available in the FINN-HLS repository. See the code snippet below taken from `StreamingMaxPool_Precision_Flexible` as an example where `NumChannels` give the worse-case bound and `NumChannels_current` gives the runtime controllable bound:

```
ActType buf[ImgDim / PoolDim][NumChannels];
#pragma HLS ARRAY_PARTITION variable=buf complete dim=2
  for(unsigned int i = 0; i < ImgDim / PoolDim; i++) {
    for(unsigned int ch = 0; ch<NumChannels; ch++){ //HW synthesized to the worse-case
#pragma HLS UNROLL
      if(ch < NumChannels_current) { // Variable loop bound
        buf[i][ch] = min_value;
      } else {
        break;
      }
    }
  }
```

F.1.2 Configuration

To make FINN automatically instantiate flexible modules, we added the `default_adaptable_mode` parameter to the FINN builder in src/finn/builder/build_dataflow_config.py (so any user can easily synthesize a runtime adaptable FINN). Setting it to `True`, informs the the following transformations (from finn/transformation/fpgadataflow/convert_to_hls_layers.py) to use the modified HLS templates: `InferBinaryStreamingFCLayer()`, `InferQuantizedStreamingFCLayer()`, `InferConvInpGen()`, `InferStreamingMaxPool()`, and `InsertDWC()`.

F.1.3 Synthesis and Evaluation

The resulting FINN accelerator, besides the traditional input/outputs (for image, weights, and classification vector), has all the _current signals exposed in the interface. In our experiments, we control these signals form the testbench (as in any other interface signal) inside the `run_rtlsim()` procedure in src/core/rtlsim_exec.py from the finn-base repository. This modification enabled us to evaluate the accelerator performance under every pruning rate. We point out that those signals could be easily controlled by a dedicated AXI interface accessible by the firmware running on the co-processor.

F.2 For the early-exit CNNs

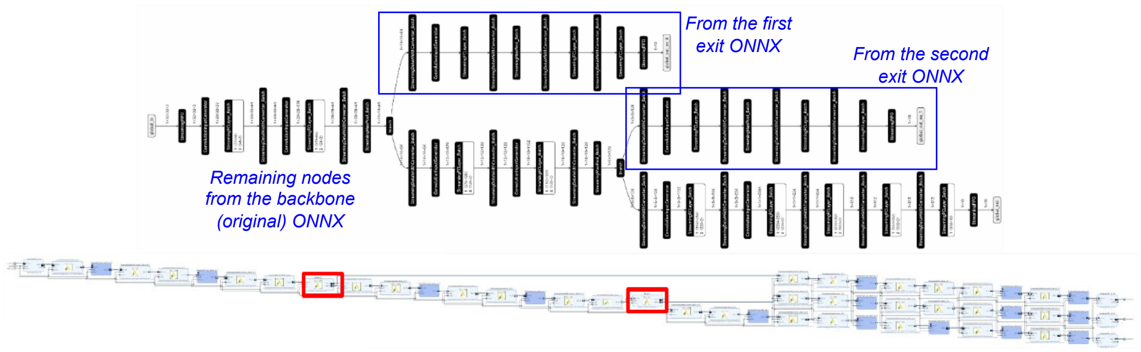

From the first exit ONNX
From the second exit ONNX
Remaining nodes from the backbone (original) ONNX

Figure F.1 – An ONNX graph with two early exits on top. The resulting accelerator (Vivado IP view) on the bottom (branch modules highlighted).

The approach taken here to compile CNNs with multiple branches was to export each branch as a single ONNX and combine them as a single model/accelerator as a FINN transformation (see Figure F.1). Doing that saved us from changing the ONNX library, keeping all our modification in the FINN compiler.

F.2.1 Setting up

In src/finn/builder/build_dataflow.py: ONNX files are passed to `build_dataflow_cfg` and JSON configuration files are read accordingly. There are one set of transformation steps that goes from `tidy up` to `step_set_fifo_depths` for the early exits (see `default_build_early_exit_dataflow_steps` in build_dataflow_config.py) and one for the backbone ONNX (see `default_backbone_data-`

`flow_steps` in build_dataflow_config.py). The transformation step for the backbone is the same as the default one, but it includes our `step_merge_backbone_exits` step, where exits are attached. Additionally, in src/finn/transformation/general.py a modified `GiveUniqueNodeNames()` adds suffixes to all model nodes in order to identify their exit number (so we do not have multiple nodes with same name and to make it possible to read the correct configuration from the JSON files).

F.2.2 Attaching Exits

In src/finn/builder/build_dataflow_steps.py: when compiling the backbone model, the `step_merge_backbone_exits` transformation is called passing the last generated ONNX intermediate files from the exits (e.g., 9_step_set_fifo_depths_ee1.onnx).

This transformation resides in src/finn/transformation/fpgadataflow/merge_backbone_exits.py. In general lines, it walks the backbone model looking for MaxPool layers (which we use as branching points in our example). When it finds one, it will add an exit there by calling `add_node_to_backbone()`. There, the necessary tensors are created to add the branch node and connect both paths. For that, we added a custom operator in src/finn/custom_op/fpgadataflow/branch.py with the finn-hlslib/branch.hpp HLS module below:

```
template<unsigned int depth, unsigned int width>
void StreamingBranch(stream<ap_uint<width> > & in,
        stream<ap_uint<width> > & out_A,
        stream<ap_uint<width> > & out_B) {
#pragma HLS DATAFLOW

 for (unsigned int y = 0; y < depth; y++) {
 #pragma HLS PIPELINE II=1
     ap_uint<width> word = in.read();
     out_B.write(word);
     out_A.write(word);
   }

}
```

Since all the steps for generating the IPs have already been called for the current model, we still need to do that for the branch that was just added. That is done by calling `PrepareIP()`, `HLSSynthIP()`, and `ReplaceVerilogRelPaths()` transformations for the remaining modules only.

Then, it is just a matter of connecting the remaining nodes from that exit. This is done by re-defining the nodes' inputs and outputs, and inserting them in the backbone model. At the end, a new global output is appended to the graph. The process repeats for all exits.

F.2.2.1 Other FINN Transformations

Additionally to the modifications presented above, some other new FINN transformations and modifications were also required to implement the exits on the accelerator. Most of the FINN modules (including the MVTU) use the `NHWC` data format, where `N` is the batch size, `H` the height, `W` the width, and `C` the number of channels. However, given the DNN topology (i.e., the input ONNX) some operators output feature maps in the `NCHW` (which, for example, is more common in PyTorch). Especially concerning our experiments, the MaxPool layer outputs in `NCHW` format. Therefore, FINN has transformations available to treat it and *transpose* the feature map when needed (e.g., when a convolutional layer follows a MaxPool, a `Transpose` is added between the two to adapt the feature map).

As for the version v0.5b FINN, the existing *MakeMaxPoolNHWC* transformation treats only the case of MaxPool with a consumer node of the type Transpose (making the sequence of modules and data formats to look like: any FINN module (NHWC) -> MaxPool (NCHW) -> Transpose (NHWC) -> any other FINN module expecting NHWC input. Therefore, a new transformation called *MakeMaxPoolNHWC_new* was added to FINN in src/finn/transformation/streamline/reorder.py. The new code adds to the existing *MakeMaxPoolNHWC* two new cases, to treat a MaxPool proceeded by Transpose and two MaxPool back-to-back. The former is used, for example, in the exits that have a MaxPool and Fully-Connected pair of layers and the latter is used when two MaxPool modules are connected (to drastically reduce the feature map size).

F.2.3 Synthesis and Evaluation

After all exits are added, we have a single model that can be synthesized with the originals `step_create_stitched_ip()` and `step_out_of_context_synthesis()` transformations (with the small modification of using the exit suffixes in the nodes' names). For our RTL simulations, we have modified the `step_measure_rtlsim_performance()` transformation. Now, it also collects data from the early exits by calling a modified `throughput_test_rtlsim()` in src/core/throughput_test.py from the finn-base repository, which in turn, runs our modified `rtlsim_exec()` in src/core/rtlsim_exec.py and, then, calls `_run_rtlsim_EE()` to read the results and count cycles from the exits and backbone: `m_axis_0_tdata`, `m_axis_1_tdata`, and `m_axis_2_tdata`, in our case. From those, we gather latency and throughput of each exit. Here, in case of execution with a board, the (early) output specified in the `-outputfile` would be dequantized and used to calculate the softmax to be accepted or not as a result (this step, however, was not implemented, but could be done in the src/finn/qnn-data/templates/driver/driver_base.py, for example).

F.3 For the split CNNs

The approach for splitting the dataflow follows the previous optimizations. It relies on two options set in FINN build configuration for setting the *split location* (after which CNN layer to split) and which side to keep (from layer one to split or from split to the last layer). The split step takes place after the `step_merge_backbone_exits`. In this way, the split can be performed over early-exit CNNs if needed and, most importantly, in this FINN step, all HLS modules have been configured, all FIFOs sized, and the dataflow is ready to be compiled by Vivado HLS and exported as an IP.

F.3.1 Setting up

To make the split in FINN configurable, we added the `split_location` and `split_for_head` parameters to the FINN builder in src/finn/builder/build_dataflow-_config.py. The first parameter sets the index layer for splitting and second flags wheter the split is intended for the *head* (initial layers) or *tail* part.

F.3.2 Splitting

A single transformation is in charge of the split: the `step_split_layers`. The transformation is straightforward, it goes over the dataflow nodes counting the number of MVTU modules and comparing it against the split location informed. When it gets to the split location, it has two possible options:

A) In case of keeping the foremost set of layers (splitting for head), it takes the output of the node at the split location and make it a global output:

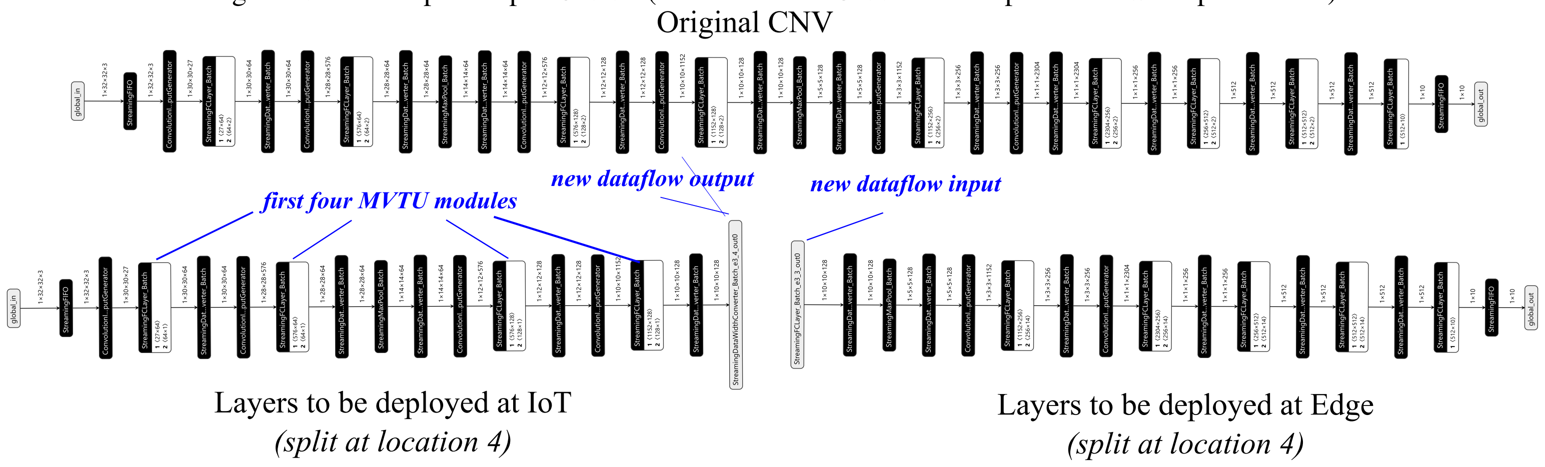
```
shape = model.get_tensor_shape(out_name)
o_lo = model.get_tensor_layout(out_name)
o_dt = model.get_tensor_datatype(model.graph.output[0].name)
global_out_new = oh.make_tensor_value_info(
    out_name, TensorProto.FLOAT, shape
)
model.set_tensor_datatype(out_name[0], o_dt)
model.set_tensor_layout(out_name[0], o_lo)
model.graph.output.append(global_out_new)
```

where *out_name* is the name of the output at split location.

B) When keeping the tail layers (from the split location up the last layer), the input of the module consuming from the module at the split location is set as the new global input, by simply *model.graph.input.append(model.get_tensor_valueinfo(node.output))*.

After adding the new global in/outs, the remaining layers are removed from the dataflow by calling *model.graph.node.remove()* on all nodes after the split if keeping the head layers or all nodes before the split otherwise. Figure F.2 shows an example of the CNV used in the experiments with a split location set at four, meaning four layers to be deployed at the IoT and the remaining five at the edge (original CNV has nine layer, top of the figure). We note also that the layers' peripheral modules (e.g., data width converters, input generators, etc) are kept with their respective MVTU modules.

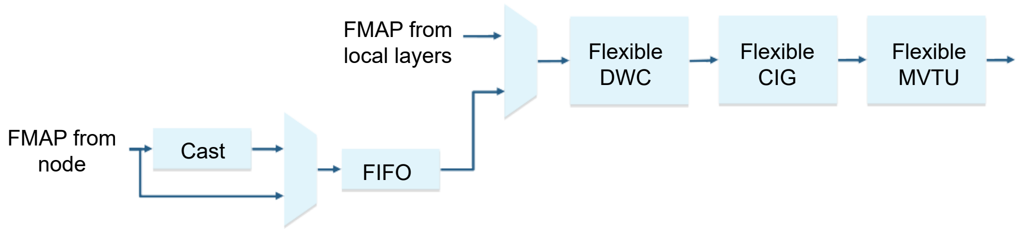
Figure F.2 – Example of split ONNX (taken from the CNV in our experiments with split set to 4)



F.3.3 FAL for dynamic depths and activation widths

The Feature Adapter Layers (FAL) proposed in the last workfront (AOI, Section 4.5) were developed to be deployed within the accelerator at the edge server. In the evaluations IoT with a runtime pruning rate were evaluated. However, it is also possible to have IoT devices changing the quantization level as well as the pruning rate. Such an approach would cause feature maps (fmaps) arriving at the FALs with chaining depth (number of channels, due to pruning) and changing bit-width (due to quantization). Although not fully integrated in the FINN design flow and, thus, not included in the main text, a module for providing a FAL capable of dealing with multiple quantization levels was investigated.

Figure F.3 – Architecture proposed for the fmap adapter.



To treat these multiple bit-widths, we developed a special "cast" for the FINN HLS modules to adapt the bit-width at runtime. The resulting hardware for the feature map adapter - including a caster module - looks like the one in Figure F.3, where the flexible DWC (Data Width Converter), CIG (Convolutional Input Generator), MVTU (Matrix-Vector Threshold Unit) are the FINN modules that were made flexible in Section 4.2 workfront. The Cast module converts the bit-width to the the width expected at the next HLS module (i.e., Flexible DWC). There are two possible cases for casting the fmap values. First, the more trivial casting from integer to integer (e.g., padding from 2 to 8 bit integers). Second, for 1-bit quantized fmaps, the Cast module will have to treat the 1-bit encoding where a bit '0' encodes a -1. Combined the flexible modules and the cast will enable the inference on feature maps from 'any' IoT configuration (i.e., at any pruning rate and quantization aggressiveness).

APPENDIX G — SOFTWARE FOR PRUNING, EARLY-EXIT, AND OFFLOADING

In this work, we implemented the CNNs in PyTorch (models used with the single-engine accelerator in the beginning of this thesis) and Brevitas, a PyTorch extension for Quantization-Aware Training developed by AMD/Xilinx (with strong support for the FINN design-flow). This appendix focuses on the Brevitas implementation since the models used in PyTorch for the single-engine accelerators are simply the original models available in the official model library. Next, we give the relevant code snippets of pruning and early-exit optimizations. Both implementations were inserted in the existing codebase from Xilinx in https://github.com/Xilinx/finn-examples under the `bnn-pynq` example that holds the CNV model (used across our experiments). Under bnn_pynq_train.py, we added options to prune, add early exits, split, and (re-)train those models. We also added all the necessary code to use the GTSRB and Caltech-256 datasets (that are not part of the Xilinx examples).

G.1 For Pruning

For implementing pruning, we followed the (LI et al., 2017) approach for pruning filters (see the "Dataflow-Aware Pruning" in Section 4.2 and "Uniform Pruning" in Appendix A). After finding the number of channels to be pruned in each layer, filters are ranked based on their $l1$-norm (`sum_of_filter`):

```
sum_of_filter = torch.sum(torch.abs(filter.view(filter.size(0), -1)), dim=1)
vals, args = torch.sort(sum_of_filter)
return args[:num_elimination].tolist()
```

Then, based on this based on this ranking, a new Brevitas layer is created (`new_conv`) keeping only the not-pruned filters (`index_remove()`) from the original one (`conv`):

```
new_conv = QuantConv2d(kernel_size=conv.kernel_size,
    in_channels=int(conv.in_channels - len(channel_index)),
    out_channels=conv.out_channels,
    bias=False,
    weight_quant=CommonWeightQuant,
    weight_bit_width=weight_bit_width)

new_weight = index_remove(conv.weight.data, dim, channel_index, independent_prune_flag)
new_conv.weight.data = new_weight
new_conv.bias.data = conv.bias.data
```

Filters are removed in `index_remove()` with:

```
new_size = tensor.size(dim) - len(index)
```

```
size_[dim] = new_size
// dim is the dimension of the feature map to be reduced
// 0 or 1 for ouput or input channels
new_size = size_
select_index = list(set(range(tensor.size(dim))) - set(index))
new_tensor = torch.index_select(tensor, dim, torch.tensor(select_index))
return new_tensor
```

After those steps, the pruned model can be retrained with the standard PyTorch/Brevitas training procedure.

G.2 For Early-exit

The early exits are appended to the CNN models in Brevitas as additional `ModuleLists()` (i.e., a list of `QuantConv2d()`, `MaxPool2d()`, `QuantLinear()`, etc. kept as an attribute in the model class). Then, in the forward call at training time all branches are available and return a classification:

```
(outputs,ees) = self.model(input)
losses_raw = [self.criterion(output,target_var) for output in outputs]
losses = [weighting * losses_raw
            for weighting, losses_raw in zip(self.model.exit_loss_weights,losses_raw)]
self.optimizer.zero_grad()
for loss in losses[:-1]:
    loss.backward(retain_graph=True)
losses[-1].backward()
self.optimizer.step()
```

Above, we can see that all exits train together (`backward()` & `step()`) and their weighted losses are summed (recall Joint Training Loss in Section 2.1.3.1). At inference time, the confidence is measured from the softmax:

```
pk = torch.nn.functional.softmax(output_vector, dim=-1)
top1 = torch.max(pk)
return ((top1 > exit_threshold).item(),top1.item())
```

where the highest softmax (`top1`) is compared to the `exit_threshold` to evaluate if that exit will be taken or not (i.e., whether the exit is confident enough).

G.3 For Splitting CNNs

Training the split CNNs was straightforward, requiring just a few modifications. The general approach taken was treat each possible quantization/split as one model, training them from scratch. Only after, they would be pruned and re-trained. Recall that in our experiments, the layers running on the final end of the split (edge server) are always running with the maximum bit-width (4 bits in our evaluations). So, for example, starting with IoT layers of 1 bit. The bnn_pynq_train.py script would train for 100 epochs the split at layer 1 (i.e., first layer at 1-bit quantization, the remaining at 4-bit). Next, the split 2, quantizing two layers with 2 bits, remaining at 4. And, so on. After all eight splits (in case of CNV) were trained, these models could be pruned with the code presented above (recall also that only layers before the split point are pruned). Therefore, only two small modifications were required. First, the `split_location` was added as a class parameter into the existing CNV class. Second, the method for pruning, now takes into account the model's `split_location` before pruning layers. This approach was taken in the work presented in Appendix E.

G.4 For Feature Adapter Layers

For training the Feature Adapter Layers (FAL), a new CNN class was created in Brevitas (which we cloud `CNVServer` for implementation purposes). The `CNVServer` class loads, at construction, two other pre-trained models those are the early-exit CNN (to be deployed on the IoT) and the CNN to be deployed on the edge. Weights for these two models are loaded from their `.tar` with `torch.load()` to initialize the corresponding layers in `CNVServer`. In order to make sure that the weights from these two models are frozen (Freeze & Train from Section 4.5.1.1), the Brevitas modules receiving the pre-trained weights are declared with the context-manager `@torch.no_grad()`. Additionally, before training stats, their weights are set with `.requires_grad = False`, so gradients are not computed for those.

FAL is the only not-trained module in the `CNVServer`:

```
...
self.FAL = ModuleList()
self.FAL.append(QuantConv2d(
kernel_size=kernel_fal,
in_channels=ch_from_IoT_split,
out_channels=ch_at_edge_split,
bias=False,
weight_quant=CommonWeightQuant,
weight_bit_width=self.weight_bit_width_fal))
self.FAL.append(BatchNorm2d(ch_at_edge_split, eps=1e-4))
self.FAL.append(QuantIdentity(
act_quant=CommonActQuant,
bit_width=self.act_bit_width_fal))
```

where `ch_from_IoT_split` is the number of channels in feature map arriving from the IoT model, `ch_at_edge_split` is the number of channels that the next layer in the edge CNN is expecting, and `kernel_fal` is the kernel size in the FAL layer (always the same size of the layer in the same position in the edge CNN). BatchNorm2d and QuantIdentity are required with the convolutional layer per the Brevitas specification.

Another modification in the `CNVServer` class (regarding a traditional Brevitas model), is that `CNVServer` has two 'modes' for the `forward()` call. The first one is for training, in this mode the sequence of layers to follow is *always* from IoT up to the split location, the FAL layer, and the remaining layers in the edge CNN. That means that no confidence measure is extracted. Then, since all layers, but the FAL are frozen, the `self.optimizer.step()` call will only *train* the FAL weights.

In the second `forward()` 'mode', we have the expected inference functioning, as described in 4.5.1.2. In this mode, the confidence is taken at every exit and it is measured against the *offload* and *exit* thresholds. Then, depending on the result, a different set of layers will be used: the exit layers, the FAL (plus remaining edge layers), or the reaming backbone layers in the IoT. The confidence tests follow the code implemented for the early-exits, only changing the if statement to `top1 <= offload_threshold`.

APPENDIX H — EXAMPLE OF A FOLDING CONFIGURATION FILE

For illustration purposes, helping the explanation of the FINN parameterization in the background (Section 2.2.2.2), we present below a commented FINN's **folding configuration** for the CNV model (this is default JSON file found in FINN repository and used for most of the experiments in this thesis).

```
{
  "Defaults": {},
  "Thresholding_Batch_0": { // This module quantizes the input image
    "PE": 1,
    "ram_style": "distributed"
  },
  "ConvolutionInputGenerator_0": { // Configures the SWU, performing the IM2COL for the first layer
    "SIMD": 3,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_0": { // MVTU for the first CONV layer
    "PE": 8,                    // PE for the first CONV layer
    "SIMD": 3,                  // SIMD for the first CONV layer
    "ram_style": "auto"
  },
  "ConvolutionInputGenerator_1": { // Pattern ConvolutionInputGenerator + MatrixVectorActivation
                                   // follows for the following layers,
                                   // note this and following indices after the '_'
    "SIMD": 16,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_1": {
    "PE": 16,
    "SIMD": 16,
    "ram_style": "auto"
  },
  "ConvolutionInputGenerator_2": {
    "SIMD": 16,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_2": {
    "PE": 8,
    "SIMD": 16,
    "ram_style": "auto"
  },
  "ConvolutionInputGenerator_3": {
    "SIMD": 16,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_3": {
    "PE": 8,
    "SIMD": 16,
    "ram_style": "block"
  },
  "ConvolutionInputGenerator_4": {
    "SIMD": 8,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_4": {
    "PE": 4,
    "SIMD": 8,
    "ram_style": "auto"
  },
  "ConvolutionInputGenerator_5": {
    "SIMD": 8,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_5": {
    "PE": 1,
    "SIMD": 8,
    "ram_style": "auto"
  },
  "MatrixVectorActivation_6": { // Last three MatrixVectorActivation have no input generators
                                // beacuse they are implementing FC layers
    "PE": 1,
    "SIMD": 2,
    "ram_style": "distributed"
  },
  "MatrixVectorActivation_7": {
    "PE": 2,
    "SIMD": 2,
    "ram_style": "block"
  },
  "MatrixVectorActivation_8": {
    "PE": 5,
    "SIMD": 1,
    "ram_style": "distributed"
  },
  "LabelSelect_Batch_0": {
    "PE": 1
  }
}
```

APPENDIX I — RESUMO EXPANDIDO EM PORTUGUÊS

Este apêndice apresenta de forma resumida esta tese de doutorado, intitulada "Otimizando para Inferência Adaptativa em FPGAs: Uma Abordagem Dinâmica Multi-Nível."

I.1 Introdução

A Internet das Coisas (IoT) é composta por componentes inteligentes com a capacidade de interagir com outros dispositivos. O mercado de IoT está se expandindo amplamente: estima-se que no próximo ano, o setor de IoT terá um impacto econômico de até 6,2 trilhões de dólares, com mais de 50 bilhões de dispositivos IoT implantados em todo o mundo. Uma das principais forças por trás dessa revolução é a Inteligência Artificial, e principalmente as Redes Neurais Artificiais, apoiando aplicações na indústria automotiva, de roupas, saúde, e muitas outras. Nesse contexto, o conceito de computação de borda se torna altamente valioso, pois permite que os dados gerados por dispositivos IoT sejam processados mais perto de onde são usados, em vez de enviá-los para a nuvem. Assim, os dispositivos IoT podem redirecionar (todos ou em parte) os dados coletados para servidores de borda que estão fisicamente próximos.

Tanto nos dispositivos IoT quanto nos servidores de borda, as FPGAs vêm sendo utilizadas para acelerar o processamento, pois atendem à demanda por flexibilidade/programabilidade oferecida pelas GPUs e a eficácia proporcionada pelos ASICs. No entanto, depender apenas de melhorias físicas (de *hardware*) pode não escalar para os requisitos dessas aplicações modernas de redes neurais, que envolvem altas cargas computacionais e impressionantes volumes de dados. Portanto, as redes neurais, em seus modelos, também devem ser otimizadas para diminuir a carga de trabalho e acelerar o processamento. Ao otimizar o modelo, podemos ajustar parâmetros como o número de camadas e o tamanho de palavra para acelerar o processamento. Diferentemente de usar um acelerador maior para processamento mais rápido, no entanto, ao otimizar o modelo, trocamos a carga computacional pela qualidade (por exemplo, acurácia).

I.2 Motivação e Contribuições

Esta tese propõe uma abordagem multi-nível e dinâmica para tornar mais eficiente a execução das CNNs em um ambiente de IoT-borda provido de FPGAs. Ao nível do hardware, a aceleração pode ser alcançada otimizando os aceleradores de FPGA que executam as CNNs. Para isso, a Síntese de Alto Nível (HLS) é uma otimização que apresenta bom resultados e tem sido bem utilizada pelo estado da arte. Com HLS, é possível ajustar o código HLS dos aceleradores para fornecer circuitos otimizados (para desempenho, recursos, energia, etc.) em FPGA.

Com relação às otimizações no modelo CNN, por outro lado, a quantização, a poda e a *early-exit* têm se mostrado alternativas proeminentes. Finalmente, além dessas otimizações no nível da CNN ou do FPGA, também devemos considerar onde a inferência é processada. Com o *offloading*, o processamento da inferência pode ser equilibrado e compartilhado entre múltiplos dispositivos no contínuo IoT-borda, multiplicando as opções de projeto (por exemplo, processando apenas algumas das camadas localmente). Portanto, o paradigma IoT-borda nos apresenta muitas oportunidades de otimização, quanto à infraestrutura de execução (por exemplo, quanto a FPGA e a localização) e quanto às CNNs em si para trocar a carga computacional pela qualidade do processamento.

Nesta tese, argumentamos que todos os eixos mencionados devem ser considerados e otimizados simultaneamente, pois se influenciam mutuamente e podem contribuir para uma inferência de maior eficiência e qualidade para o usuário final. A contribuição geral desta proposta de tese reside na otimização das CNNs para execução em FPGAs no contínuo IoT-borda, abordando simultaneamente o modelo CNN e sua implementação em hardware. E, além disso, propomos explorar esses modelos e aceleradores otimizados de forma dinâmica e adaptativa, dado que os cenários modernos de IoT-borda são altamente heterogêneos, com múltiplas aplicações concorrentes, cargas de trabalho variáveis e ambientes diversos.

I.3 Resultados Principais

Ao decorrer desta tese de doutorado, foram avaliados aceleradores FPGA do tipo *single-engine* e *dataflow*, três arquiteturas de CNN, otimizações de computação aproximativa, poda, quantização, *early-exit*, *offloading* e inferência partida (*split*). O resultado principal atingido nesta tese é de que, quando usadas de forma inteligente, essas otimizações podem ser combinadas para prover ganhos em desempenho, consumo energético, e eficiência. E, mais importante, essas otimizações podem ser utilizadas para adaptar o funcionamento em tempo de execução. Ao final da tese, mostramos a ferramenta chamada AOI que realiza a combinação e a exploração propostos. Ao comparar a inferência otimizada com o AOI contra inferências estado da arte (executadas por aceleradores *open-source*) em placas FGPA de IoT e borda, observamos os seguintes ganhos.

Quanto à acurácia das inferências, notamos relativo aumento no AOI, mesmo quando fortemente podado. Os ganhos em acurácia ficam evidentes no dispositivo IoT quando observado a acurácia entregue pelo AOI e queda da acurácia, sob os mesmos níveis de poda, da CNN estado da arte. Os bons níveis de acurácia do AOI se devem à (i) divisão da inferência entre as CNNs de IoT e de borda, no qual camadas de maior capacidade são executadas na borda sem custo para o dispositivo IoT; e (ii) ao uso da otimização de early-exit no IoT que, entre outras vantagens, torna as CNNs mais resilientes a poda (i.e., apresentam queda de acurácia menos acentuada).

Quanto ao desempenho na execução das inferências, observamos primeiramente que, quando sob conexão volátil entre IoT e borda (e.g., comunicação em rede de telefonia), a execução em uma solução fixa (i.e., que não adapta) é inviável. Ao comparar a latência provida no AOI contra a provida pelo estado da arte - mesmo quando este foi otimizado, ficou claro que o processamento de inferência estático, mesmo que otimizado, pode fornecer baixa latência *ou* alta acurácia, mas dificilmente cobre todos os comportamentos necessários em tempo de execução. Portanto, ao explorar as otimizações de forma conjunta e adaptativa, AOI acelera o processamento das inferências de forma consistente (sob todas condições de comunicação IoT-borda).

Em comparação com o acelerador estado da arte executando no IoT ou no servidor de borda, o AOI oferece melhores níveis de Perda de Quadros (*frame loss*). Graças à adaptação em tempo de execução do AOI, pode-se ajustar os parâmetros da taxa de poda, da *early-exit* e do *offloading* de acordo com as condições atuais de rede de comunicação. Como resultado, o AOI equilibra o alto rendimento do servidor de borda (retornando perda mínima de inferência) e a baixa latência fornecida pela FPGA local no IoT.

Quanto à eficiência energética, equilíbrio encontrado pelo AOI entre maior vazão da borda e menor latência local, também faz com que o AOI entregue a melhor eficiência energética. Mesmo utilizando FPGAs nos dois extremos, IoT e borda, o AOI é 1,60x e 3,9X mais eficiente (em termos de número de inferências por Watt) do que aceleradores de estado da arte rodando nas mesmas plataformas.