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

JULIO COSTELLA VICENZI

Exploiting Virtual Layers and Reconfigurability for FPGA Convolutional Neural Network Accelerators

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microeletronics

Advisor: Prof. Dr. Antonio Carlos Schneider Beck Filho
Coadvisor: Prof. Dr. Mateus Beck Rutzig

Porto Alegre
November 2023

CIP — CATALOGING-IN-PUBLICATION

Costella Vicenzi, Julio

Exploiting Virtual Layers and Reconfigurability for FPGA Convolutional Neural Network Accelerators / Julio Costella Vicenzi. – Porto Alegre: PGMICRO da UFRGS, 2023.

78 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2023. Advisor: Antonio Carlos Schneider Beck Filho; Coadvisor: Mateus Beck Rutzig.

1. FPGA. 2. Deep neural networks. 3. Hardware accelerators. 4. Pruning. 5. Quantization. I. Schneider Beck Filho, Antonio Carlos. II. Beck Rutzig, Mateus. III. Título.

To learn which questions are unanswerable, and not to answer them:
this skill is most needful in times of stress and darkness
— URSULA K. LE GUIN, THE LEFT HAND OF DARKNESS

AGRADECIMENTOS

A realização deste trabalho contou com a colaboração de diversas pessoas, às quais merecem os devidos agradecimentos.

Primeiramente, expresso minha gratidão aos meus pais, Nivaldo Luis Vicenzi e Marinez Costella (in memoriam), pelo amor, carinho e apoio incondicional ao longo de toda a minha jornada. Aos meus tios Juarez Rode e Elenice Rode, agradeço pelo apoio desde o início da minha graduação e durante a pesquisa na pós-graduação.

Agradeço a todos os meus amigos que estiveram ao meu lado em todos os momentos, com destaque para Bruno Bertoldi, Gert Folz, Priscila Poletti, Barbara Donida e, por fim, Gabriel Debona, que me acompanhou desde cedo e me acolheu durante a mudança para Porto Alegre.

Ao meu orientador neste trabalho, Professor Antonio Carlos Schneider Beck Filho, agradeço pela paciência, conselhos e ensinamentos compartilhados. Agradeço também ao meu coorientador, Professor Mateus Beck Rutzig, pela significativa contribuição na realização deste trabalho e por seu papel fundamental na fomentação do meu conhecimento de pesquisa científica desde a graduação.

Agradeço a todos os colegas de laboratório, com destaque para Guilherme Korol e Michael Guilherme Jordan, cujas contribuições foram fundamentais, tanto do ponto de vista técnico quanto científico, para a elaboração deste e de outros trabalhos.

Expresso minha gratidão à CAPES pelo auxílio financeiro e também aos professores e funcionários do Programa de Pós-Graduação em Microeletrônica e do Instituto de Informática, que proporcionam as condições para o ensino de excelência oferecido pela Universidade Federal do Rio Grande do Sul.

ABSTRACT

Artificial neural networks (ANN) are a solution for many classification problems, from face recognition to malware detection in computer network packets. With great accuracy, ANN algorithms are computationally expansive, requiring millions of arithmetic operations for a single classification inference. To overcome this challenge, hardware accelerators such as graphical processing units (GPUs) or field-programmable gate arrays (FPGAs) are implemented to reduce inference time and energy consumption. FPGAs have two standout characteristics: reconfigurable architecture and low-power capabilities, allowing for tailor-made designs for any ANN, while maintaining flexibility of functionality as requirements change over time. This work explores the case study of a smart Network Interface Card coupled with an FPGA accelerator required to implement four distinct ANNs, taking computer network packets as input. As the bandwidth in a computer network changes over time, the throughput requirements for the FPGA accelerator also vary. Three optimization frameworks to exploit the reconfigurable nature of FPGAs, focusing on maximize the quality of experience (QoE) and minimize energy consumption. Anya is a framework that dynamically reconfigures the FPGA accelerator according to the current classification task and the number of incoming inference requests. From a library of pruned accelerator designs, Anya leverages the trade-off between accuracy versus throughput, exploring the design space of pruning over time, assuring that the highest number of inferences are performed with optimal accuracy while managing multiple classification tasks. Hardware Virtual Layers Tara employs hardware virtual layers, aiming at reducing the number of FPGA reconfigurations required to change between classification tasks, implementing a custom hardware architecture. An added virtual hardware layer is implemented in each accelerator design, allowing for a single tailor-made accelerator to implement multiple tasks seamlessly. Spyke combines these approaches, using a set of pruned hardware accelerators using hardware virtual layers, following Anya's dynamic framework. Experiments with four different scenarios show that Spyke increases the QoE by up to $1.13\times$ and reduces the energy per inference by up to $1.40\times$. Tara also presents up to $1.13\times$ increase in QoE while not requiring any reconfigurations. The best results are found from Spyke, with an increase in $1.22\times$ in QoE and $1.37\times$ more processed frames with a reduction of up to $1.35\times$ energy.

Keywords: FPGA. deep neural networks. hardware accelerators. pruning. quantization.

Exploração de Camadas Virtuais e Reconfigurabilidade para Aceleradores de Redes Neurais Convolucionais em FPGA

RESUMO

As redes neurais artificiais (ANN) são uma solução para muitos problemas de classificação, desde o reconhecimento facial até a detecção de malware em pacotes de redes de computadores. Com grande acurácia, os algoritmos ANN são computacionalmente custosos, exigindo milhões de operações aritméticas para uma única inferência de classificação. Para superar esse desafio, os aceleradores de hardware, como as unidades de processamento gráfico (GPUs) ou FPGAs (field-programmable gate arrays) são implementados para reduzir o tempo de inferência e o consumo de energia. Os FPGAs têm duas características de destaque: arquitetura reconfigurável e recursos de baixo consumo de energia, permitindo projetos personalizados para qualquer ANN, mantendo a flexibilidade da funcionalidade à medida que os requisitos mudam com o tempo. Este trabalho explora o estudo de caso de uma placa de interface de rede inteligente (SmartNIC) acoplada a um acelerador FPGA para implementar quatro ANNs distintas, tendo como entrada pacotes de rede. Como a largura de banda em uma rede de computadores muda com o tempo, os requisitos de processamento do acelerador FPGA também variam. Três *frameworks* de otimização para explorar a natureza reconfigurável dos FPGAs são propostos, com foco em maximizar a qualidade da experiência (QoE) e minimizar o consumo de energia. Anya é uma estrutura que reconfigura dinamicamente o acelerador de FPGA de acordo com a tarefa de classificação atual e o número de solicitações de inferência recebidas. A partir de uma biblioteca de designs de aceleradores podados, Anya aproveita a compensação entre precisão e taxa de transferência, explorando o espaço de design da poda ao longo do tempo, assegurando que o maior número de inferências seja realizado com a precisão ideal enquanto gerencia várias tarefas de classificação. Tara utiliza camadas virtuais de hardware com o objetivo de reduzir o número de reconfigurações de FPGA necessárias para alternar entre as tarefas de classificação, implementando uma arquitetura de hardware personalizada. Uma camada de hardware virtual adicional é implementada em cada modelo de acelerador, permitindo que um único acelerador feito sob medida implemente várias tarefas sem problemas. Spyke combina essas abordagens, usando um conjunto de aceleradores de hardware podados usando camadas virtuais de hardware, seguindo a estrutura dinâmica da Anya. Experimentos com quatro cenários diferentes mostram que

Anya aumenta a QoE em até $1,13\times$ reduz a energia por inferência em até $1,40\times$. Tara também apresenta um aumento de até $1,13\times$ na QoE, sem exigir nenhuma reconfiguração. Os melhores resultados foram obtidos com a Spyke, com um aumento de $1,22\times$ na QoE e $1,37\times$ mais quadros processados com uma redução de até $1,35\times$ na energia.

Palavras-chave: FPGA. redes neurais profundas. aceleradores de hardware. poda. quantização.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
DSE	Design Space Exploration
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
CONV	Convolutional layer
FC	Fully connected layer
PR	Pruning rate
PRR	Partial Reconfiguration Region
SR	Static Region
DPR	Dynamic Partial Reconfiguration
I/O	Input/output

CONTENTS

1 INTRODUCTION

In computer network monitoring, traffic classification is a crucial task that extracts relevant features, such as the type of application used, protocols, and encryption (CISCO, 2023). These features provide an understanding of how the network resources are used and can also aid in security monitoring for anomaly and intrusion detection, providing information on potential network attacks and improper use of network resources. They are also the foundation for many quality of service (QoS) modules and features in modern network infrastructure. For this, it is possible for routers and switches to fast-forward or not drop packets that fit certain classification criteria.

Identifying traffic is not trivial, especially in systems with encrypted packets (CAO et al., 2014). Thus, Deep neural networks (DNNs) have been extensively employed for these classification problems, as they can be trained to identify different classification features required for monitoring, providing a solution to many problems. Multi-Layer Perceptron (MLP) networks have been deployed in this context, and more recently, Convolutional Neural Networks (CNNs) have shown great promise, providing highly accurate classifications at the cost of a heavy computational burden (WANG et al., 2017a).

Although effective at accurately classifying packets, DNNs require powerful computation systems to provide real-time inferences. Moreover, internet speeds are growing each year, with newer network infrastructure supporting port speeds of over 100 Gbps. This speed increase leads to an ever greater number of packets flowing through the network that require classification, leading to systems requiring a processor able to perform many times more inferences per second than their maximum throughput. Therefore, using the correct hardware solution is essential to handle this workload.

A simple network interface card provides the physical interface for network communications via a simple processor. As the complexity of computer networks grows, more functions and tasks are required of these devices, leading to the use of more robust hardware solutions. For instance, much of the computing is dedicated exclusively to communication in modern data centers. Smart Network Interface Cards (SmartNICs) provide a platform that helps to overcome these challenges. These special network cards offer highly specialized accelerator units that make network management, security, and storage more efficient and robust. These accelerators can process some or all of the workload previously handled by the CPU contained in the NIC, allowing for parallel tasks to be executed faster while freeing the CPU for more appropriate management tasks. Through

the use of Software-Defined Networks (SDN) and integrated accelerators such as Field-Programmable Gate Arrays (FPGA) (LOCKWOOD et al., 2007), SmartNICs allow for the benefit of fully custom and flexible accelerators. Especially for such *reconfigurable* SmartNICs, the flexibility provided by reconfiguring the FPGA with new accelerators allows multiple custom designs to be explored on demand.

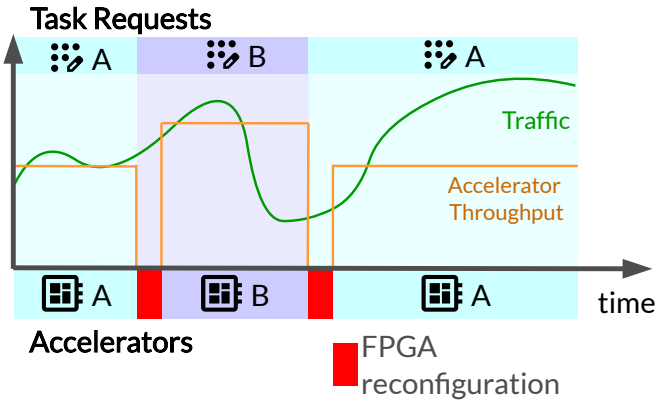
FPGAs offer benefits over ASIC or GPU accelerators (NURVITADHI et al., 2017). They consume almost one order of magnitude less power than GPUs, allowing for energy-efficient accelerator designs. Their adaptability also allows for different tasks to be deployed over time as required, expanding the number of metrics the network can monitor while offering designs tailor-made for each classifier to fit the application and the user's needs, unlike ASIC's generic or fixed application. One drawback of FPGAs is their reconfiguration time, leading to challenges in dynamic reconfiguration for time-critical tasks.

Clever accelerator designs can mitigate the overhead of using FPGA reconfiguration. Via virtual layers (detailed later in this dissertation), it's possible to fit an accelerator that maintains the application-specific design of FPGA accelerators while allowing for various FPGA tasks to be executed and switched without any performance loss. Reducing time spent on overhead operations gives the system extra flexibility and a more complete picture of the flowing traffic.

On top of that, even though reconfigurable SmartNICs can improve the efficiency of CNN processing, *relying exclusively on the hardware cannot sustain performance and energy gains indefinitely* - especially in a heavy-load context of large CNNs and high traffic volumes. In this context, the CNN models must be optimized as well. Two main techniques are used in this work to maximize the throughput of a model: quantization and pruning.

Since CNNs are usually normalized to a restricted number range around zero, many of the values represented by floating point numbers are redundant and give space for compression. Quantization reduces the precision of CNN parameters (weights, biases, and activations) to lower numeric precisions, from half-precision floating point to integers. This optimization can lead to a small reduction in classification accuracy, but provide faster arithmetic operations with fewer bits and a lower memory footprint required to store CNN parameters (PAPPALARDO, 2021). Pruning is a popular technique to reduce the CNN size and computation requirements at the cost of accuracy. Pruning removes entire parts of a CNN, from neurons to entire filters, reducing the number of parameters required to be stored in memory and the number of calculations performed in each inference (LI et

Figure 1.1: Example of traffic and task requests over time.



Source: the author.

al., 2017).

1.1 Challenge

Using SmartNICs coupled with FPGAs to deploy multiple neural network classification tasks is difficult due to the number of inferences required to classify incoming traffic. This performance requirement forces the design of FPGA accelerators, which are a crucial design point for the system, as they must meet the processing speed requirements while also maintaining great classification accuracy, low power, and energy consumption, and make the most of the hardware's resources. Moreover, the ability to adequately switch between classification tasks over time with the minimum overhead costs also requires further optimizations.

To more clearly illustrate the problem, Figure 1.1 shows the SmartNIC sending the inference requests to the FPGA. Different classification tasks are requested over time in a system composed of a SmartNIC coupled with an FPGA. These tasks are processed via accelerators running on the FPGA. As the different tasks are requested, the system must reconfigure the FPGA with the corresponding accelerator. On top of the graph, two different classification tasks A and B, are displayed as requested over time. The x-axis shows the time, and the y-axis indicates a green curve for the accelerator throughput and a blue one for the incoming traffic. At the bottom of the graph, the accelerators configured on the FPGA are shown, with reconfiguration time overhead shown in red. The time required to reconfigure an FPGA device varies according to multiple factors, such as FPGA part, disk I/O, PCIe latency, bitmap size, and many other overheads (Xilinx, 2020; XILINX, 2019; MOODY, 2021), ranging from a few hundreds of milliseconds to over

a second. As the number of requests over time is sometimes higher than the processing capabilities of the accelerator, some packets are lost. This leads to worse metrics for network management, as the missed packets will not be classified.

The main focus of this work is to exploit the reconfigurability of the FPGA to allow for more inferences to be performed, while still employing multiple classifications tasks over time. Using dynamic accelerator designs tailored to the NN architectures can lead to great performance, but comes at the trade-off of reconfiguration times and energy spent, where the device is not performing inferences.

1.2 Contributions

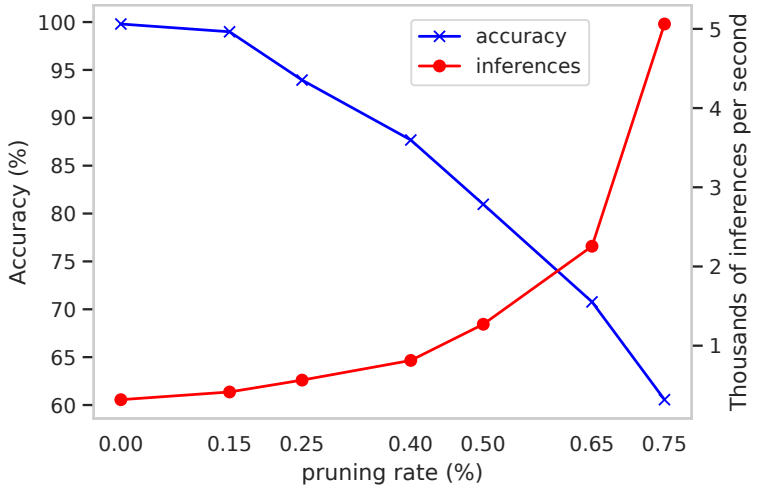
To enable the smartNIC to maximize the number of inferred packets while maintaining the flexibility of reconfigurable hardware, this work proposes two solutions. In order to increase the number of inferences per second, a framework using a library of pruned accelerators, and a virtual hardware layer technique to create accelerators capable of performing multiple classification tasks, reducing the FPGA's reconfiguration overhead.

The framework is composed of a library of pruned accelerators which are reconfigured dynamically. Different accelerators are reconfigured to the FPGA according to the incoming network traffic and the classification task. Figure 1.2 shows an example of the performance *versus* accuracy trade-off created by pruning. As the pruning rate increases, the accuracy drops while the throughput dramatically increases. Thus, pruning fits well with the network's traffic fluctuations. By reconfiguring the SmartNIC, it is possible to achieve faster inferences when the network is overloaded or higher accuracy levels whenever the network is experiencing slower traffic. On top of that, such flexibility also allows switching between multiple traffic classification tasks at runtime.

To reduce the reconfiguration overhead of switching between classification tasks, a virtual hardware layer technique is proposed. This allows the same hardware accelerator to be used for all tasks in the system, allowing tasks to be swapped easily with no time or energy spent reconfiguring the FPGA. This reduction in overhead allows for more packets to be classified and great flexibility for network managers to gather diverse network metrics.

Given the opportunities of the trade-off granted by pruning and the need for seamless task switching, to improve the performance in these scenario, this work proposes

Figure 1.2: Accuracy and inferences per second w.r.t pruning rate.



Source: the author.

three frameworks: **Anya, Tara** and **Spike**:

- **Anya**: A framework composed of a pruned accelerator library, optimizing the FPGA accelerator configuration according to classification task requests and the incoming network traffic. It aims to optimize the service's Quality of Experience (QoE) by selecting the accelerator according to the network traffic. It employs more heavily pruned accelerators with greater throughput when the network is overloaded and higher accuracy accelerators when the traffic is slower, enabling multiple heavy-load CNNs to process traffic data in very dynamic conditions.
- **Tara**: a hardware technique to expand the generality of accelerators via hardware virtual layers (HWVL), allowing for multiple classification tasks to be switched without FPGA reconfigurations, using the same accelerator model. This allows for the dynamic switching of tasks with no time or energy overhead, with more time for inference processing instead.
- **Spike**: a combination of Tara's HWVL in Anya's accelerator library, exploiting the benefits of both proposals.

1.3 Work Structure

The remainder of this work is structured as follows: Chapter 2 presents convolutional neural networks, pruning and quantization optimization techniques, dataflow hardware accelerators, and dynamic FPGA reconfigurable accelerators. Chapter 3 presents the

State-of-the-Art works related to this dissertation. Chapter 4 details the proposed Anya, Tara and Spike frameworks. Chapter 5 details the experiments performed and evaluates the performance of the proposed solutions. Finally, Chapter 6 concludes this work, summarizing the results and discussing future research possibilities.

2 BACKGROUND

This chapter presents the theoretical background used in this work. Section 2.1 presents Machine learning and artificial neural networks, Section 2.2 discusses Convolutional Neural Networks (CNNs), Section 2.3 explores the machine learning optimization techniques utilized, Section 2.4 presents FPGAs as the hardware platform for the deployed accelerators, Section 2.4.1 presents tools used to generate hardware accelerators, Section 2.5 shows adaptable hardware accelerator designs.

2.1 Machine Learning and Artificial Neural Networks

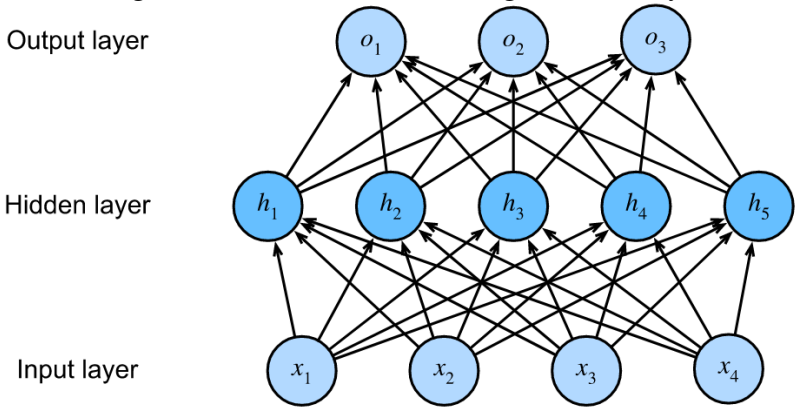
Machine learning has been widely adopted for a wide range of problems, from malware (WANG et al., 2017b) to network intrusion detection (VINAYAKUMAR; SOMAN; POORNACHANDRAN, 2017). Unlike classical algorithms, which require the programmer to define rigid rules for program output, machine learning algorithms instead learn these rules autonomously from huge sets of problem samples (ZHANG et al., 2021).

Artificial neural networks (ANN) are based on the human brain's anatomy, using similar terminology, as it aims to replicate its learning capability. These classification-based algorithms take an input (a signal, an image, sound, or even raw data) and infer which target class the input fits into. With a set of target classes and inputs, many **labeled** samples can be aggregated into a dataset. For example, a dataset of packets labeled as VPN or non-VPN is required to detect computer network packets that use VPN encryption. With a dataset for a given problem, the algorithm is trained on the data, learning the important features of the different classes. The main function of training is to use a dataset of labeled examples and find a function $f(x)$ that can be optimized via learnable parameters. These parameters are adjusted based on the CNN's prediction errors compared to the dataset's true label. This is done through a series of learning iterations called epochs, each running through all the dataset's examples.

Once the training is complete, the model is ready to predict unknown inputs (i.e., packets not used during training). This second phase is called inference and the focus of this work. The heuristic nature of machine learning can lead to classification errors, making accuracy a crucial feature to optimize during training.

The basic feature of a neural network is a neuron. Neurons receive inputs (synapses) from other neurons or as the ANN's input. Each synapse is multiplied by a respective

Figure 2.1: An ANN with a single hidden layer.



Source: Zhang et al. (2021).

weight, which is accumulated and added to a bias. They are called the parameters of a neuron, allow the ANN to generalize any function, and are tuned during training. Then an activation function is used, generating an output synapse. Activation functions add non-linearity to the equation, intended to emulate the human neuron's firing behavior if the incoming signals are greater than a certain threshold. Equation 2.1 expresses the operations of the k-th neuron with n inputs, where b is the bias, w_i and x_i are the i-th weight and inputs of the neuron with the activation function ϕ.

$$y_k = \phi(b_k + \sum_{i=0}^{n} w_{k,i} x_i) \tag{2.1}$$

A common activation function is the Rectified Linear Unit (ReLU), shown in Equation 2.2. It is also known as the ramp function, which returns zero for any negative value and the input value otherwise. It is simple to implement in computer hardware, which makes it efficient.

$$\phi(x) = max(0, x) \tag{2.2}$$

If not for the non-linear properties of the activation functions, grouped layers of neurons could be collapsed into a single linear equation, making a layered network redundant. Indeed, deep learning basis is forming groups of neurons into layers. A fully connected (FC) layer is formed from a set of neurons, chaining multiple layers together so each neuron receives all outputs from the previous layer. Thus each neuron being **fully** connected to the previous' layer output. With multiple layers in a network, each connected to the last layer's output forms a fully connected layer. Equation 2.3 shows how the computations in a layer of neurons can be represented as matrix multiplication and

vector addition. Here the activation function is applied element-wise.

$$\boldsymbol{y} = \phi(\boldsymbol{x}^T \boldsymbol{w} + \boldsymbol{b}) \tag{2.3}$$

The set of FC layers forms a neural network (NN), as shown in Figure 2.1. Feed-forward NNs with only FC layers are called multilayer perceptions (MLP). The layers between the input and output are specially called hidden layers, and ANN with multiple hidden layers are denominated deep neural networks (DNN). The number of neurons in each layer, the choice of activation function, and the number of hidden layers define the topology of a network. The greater the number of neurons and layers, the greater the memory and computations required. The advances in computer technology in recent years have allowed for ever bigger networks with millions of parameters.

2.2 Convolutional Neural Networks

Two main characteristics of MLP models are that the relations and order between inputs do not impact the resulting inference, and all inputs must be flattened into one-dimensional arrays. This means that the relations of input features are disregarded for problems involving related features or multidimensional data, such as images, time series, and network packets. For example, in an image edge detection problem, the order in which the pixels appear is fundamental, as the target feature is exactly the spacial relation between similarly colored pixels, information that would be lost by flattening the input image and not fully explored by fully connected layers.

To explore problems with related features, Convolutional Neural Networks (CNNs) employ convolutional layers before FC layers, allowing the relations between input features to be extracted before classification. They have found great success in many areas, such as image classification and traffic classification (ACETO et al., 2019) as well, as explored in this work.

Convolutional layers use trainable filters called kernels. These kernels have the same dimensions as the input and multiple channels. Filters act as "feature extractors," learning patterns from the input that will be passed into the next layers. Since the same filters are applied to the whole input, they are invariant to translations in the input, analyzing each window separately. During a convolution operation, the filter is applied via convolution operation across the input, following Equation 2.4. Here a one dimensional

input $\boldsymbol{x}$ with n_c channels is convolved with the filter $\boldsymbol{w}$ of size $2 * p$, the resulting in the j-th element of the feature map. This operation is applied to all elements of the input. Since the same kernel is applied to all inputs of this layer, their relations and variance are considered.

$$y_j = b_j + \sum_{c=1}^{n_c} \sum_{k=-p}^{p} x_{c,j+k} w_{c,k} \tag{2.4}$$

In the same way as FC layers, the convolution operation is followed by a non-linear activation function. Some models also employ pooling layers (POOL). These are used to downsample the feature map, extracting a summary of that region. These layers may implement several different techniques, such as average pooling, calculating the average value elements in the pooling kernel and max pooling, sampling the largest element. This not only reduces the dimensions of the feature map, thus reducing the number of calculations in subsequent layers, but also adds the translation invariance property to the convolution. It is what allows for extracting features regardless of their position on the feature map, such as shifting the input by a few samples or a feature located in a different location of the input feature map (GOODFELLOW; BENGIO; COURVILLE, 2016).

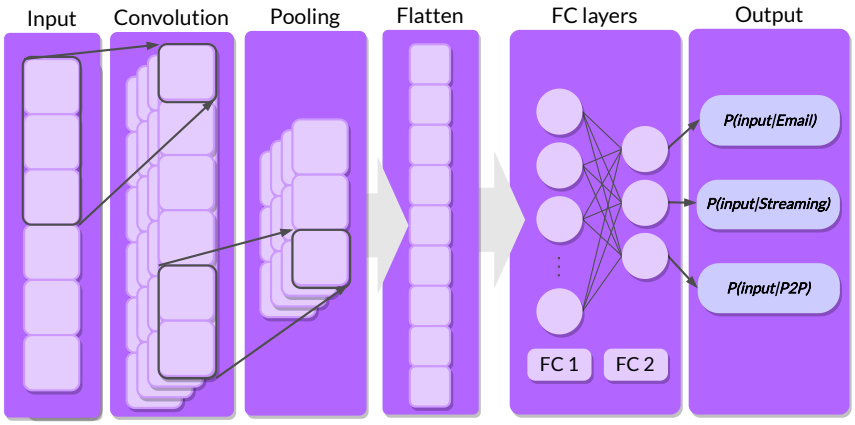
After the last CONV layer, the feature maps are flattened into 1d arrays and input into FC layers, as seen in Section 2.1. Since convolution is computationally expensive, CNNs have most of their processing in the convolutional layers, while most parameters belong to FC layers.

Figure 2.2 shows an example where the network receives a traffic sample (e.g., a computer network packet) as input and predicts what category, from a given set of problem-defined classes (i.e.: labels), it fits best. The output is a list with the probabilities of the input belonging to the respective class. In this example, three classes are shown, representing which application generated the network packet: Email, Streaming, or P2P applications.

The process of training a DNN model is difficult and lengthy. To facilitate and speed up training, batch normalization layers are a popular method, added after CONV or FC layers. The layer acts as a regularization step of the intermediate inputs between the other layers of the DNN, normalizing the inputs by their mean and standard deviation. This makes the intermediary output of each network layer more stable. Batch normalization is also important in quantization, as discussed in Section 2.3.1.

The work Zhang et al. (2019) shows the importance of efficiently employing CNN-based traffic classification systems. From all considered classification methods, CNN de-

Figure 2.2: Example of a one-dimensional convolutional neural network.



Source: the author.

livers the highest accuracy for traffic classification tasks (BOUTABA et al., 2018; ACETO et al., 2019; ZHANG et al., 2019).

2.3 DNN optimization techniques

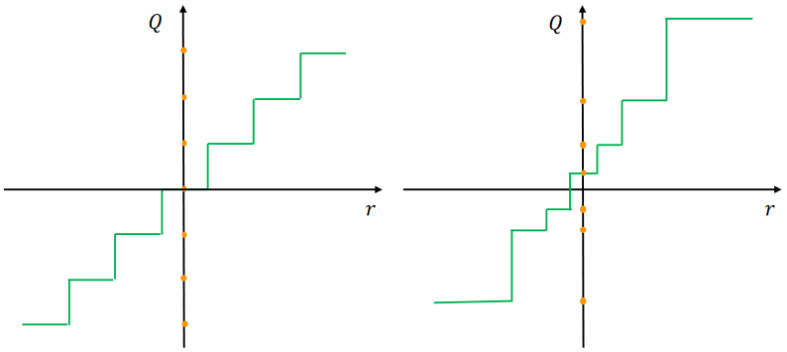
This section presents the optimization techniques used in this work: quantization and pruning.

2.3.1 Quantization

Quantization is a method of reducing the computation complexity and memory footprint of a DNN. Usually, DNNs use single (32-bit) or double (64-bit) precision floating point numbers to represent their parameters. However, the range of values represented by the IEEE 745 standard (IEEE. . . , 2019) creates redundancy, as many representable values are never actually used during computations. Since DNNs are overparametrized (i.e., the number of parameters is greater than required to achieve their accuracy), this leads to a great opportunity for optimization when employing quantization from half-precision floating point (16-bit) to binary networks (1-bit), which can reduce computations to simple logic gates (GHOLAMI et al., 2021). *All experiments in this work employ DNN quantization.*

Quantization maps the continuum real domain into a discrete, lower precision, quantized domain. Figure 2.3 shows two examples of quantization. Here the x-axis shows the real domain and the y-axis the quantized values marked with orange dots. On the left

Figure 2.3: Examples of uniform (left) and non-uniform (right) quantization.



Source: Gholami et al. (2021).

uniform quantization is applied, meaning that real values are mapped into lower precision at fixed intervals (quantization levels). This is a simple solution and easily implemented. On the right non-uniform quantization is shown with varying quantization levels, allowing specific ranges of values to have improved precision over other less relevant ranges. In this example, more quantization levels are allotted to values near zero at the expanse of bigger values. This work focuses on uniform quantization.

Equation 2.5 shows how to map a real value r with a S scaling and Z the zero point.

$$Q(r) = int(rS) + Z \tag{2.5}$$

Here $int(.)$ truncates the value to the nearest integer value, rounding down. This allows for representation of any $r \in [\beta, \alpha]$ with b-bits wide integer $Q(r) \in [-2^{b-1}, 2^{b-1} - 1]$.

The value can be dequantized via Equation 2.6.

$$\overline{r} = \frac{1}{S(Q(r) - Z)} \tag{2.6}$$

The values for scaling (S) and zero point (Z) parameters have a big impact on the performance of a DNN (JACOB et al., 2017; KRISHNAMOORTHI, 2018). Affine or asymmetric quantization utilizes a zero point different from integer zero. Equations 2.7 and 2.8 show their definition for b-bit wide values.

$$S = \frac{2^b - 1}{\alpha - \beta} \tag{2.7}$$

$$Z = -int(\beta S) - 2^{b-1} \tag{2.8}$$

Here S is a real value, while Z is constrained to an integer value, meaning that zero will not be rounded to another value, having an exact representation. This is important since operations such as Rectified Activation Unit (ReLU) directly compare to zero, alleviating any possible precision loss in the function that must be calculated hundreds of times per inference.

Further constraining $Z = 0$, a simpler scale symmetric quantization is achieved, constraining the representable range to $[-\alpha, \alpha]$. This simplifies Equation 2.5 and 2.7 to a simpler division and multiplication operations shown in Equations 2.9 and 2.10 respectively. In this case, the real zero is mapped directly to the integer zero.

$$Q(r) = int(rS) \tag{2.9}$$

$$S = \frac{2^{b-1} - 1}{\alpha} \tag{2.10}$$

The definition of α and β can be found via calibration. This takes the trained network and its training datasets, measuring the error and tuning these parameters (VANHOUCKE; SENIOR; MAO, 2011). Quantization can also be applied differently from layer to layer to decrease the precision loss with quantization.

Quantization-Aware Training (QAT) trains the DNN using reduced precision values to minimize the quantization error and accuracy loss. The forward pass emulates the values of parameters following Equation 2.5, so the loss includes the added arithmetic error caused by quantization. However, this incurs problems in the backpropagation phase of training, where the derivative is undefined at step boundaries and zero anywhere else (GHOLAMI et al., 2021). A straight-through Estimator (STE) is used to circumvent this issue (BENGIO; LÉONARD; COURVILLE, 2013), replacing the original derivative. This sets the derivative as one for values in $[\beta, \alpha]$ and zero anywhere else.

Brevitas (PAPPALARDO, 2021) is an open-source PyTorch-based QAT library developed by Xilinx. It implements quantized versions of modules (layers). The user can replicate any DNN with quantized parameters and train it easily, specifying the type of quantization and the bit-width of each parameter. The training process is the same as any regular PyTorch model, simplifying the conversion from a non-quantized network to a quantized version. All DNNs presented in this work use Brevitas for QAT.

2.3.2 Pruning

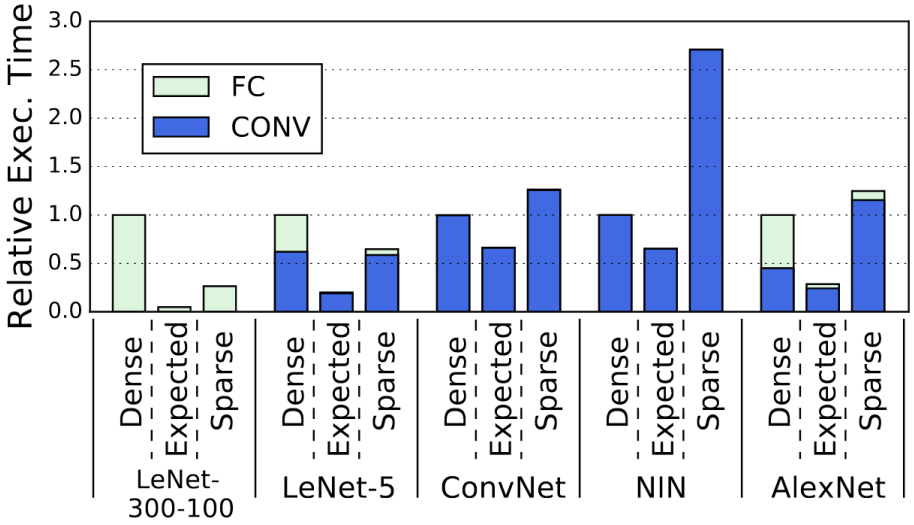
Pruning is a method to reduce the number of required computations and the number of store parameters in an overprovisioned DNN by removing neurons or entire filters (CHOUDHARY et al., 2020) It can be applied alongside other optimization methods, such as quantization.

Modern DNN models contain millions of parameters, allowing for an easier training process but adding some redundant parameters (DENIL et al., 2013). However, the size of such networks is a hindrance when they are deployed, leaving optimization space for compression. The pruning process is more efficient than training a new model from scratch, as the pruned model achieves higher accuracy with no or some retraining (LI et al., 2016b). Theoretically, a pruning algorithm is capable of removing parameters at no accuracy cost. However, this is an NP-hard problem (GUO; YAO; CHEN, 2016).

In a finer granularity, unstructured pruning, such as weight pruning, removes selected neurons from a layer, leaving the mathematical representation of a layer as a sparse matrix (RETSINAS et al., 2020). However, the computation of spare matrices can cause slowdowns (YU et al., 2017), making the use of pruning unviable. Figure 2.4 shows the execution time of the original Dense networks and the pruned Sparse networks alongside their Expected execution times based on the remaining multiply and accumulate operations required. In the case of ConvNet, NiN, and AlexNet, the pruned network takes longer than the original, even though their expected execution time is many times smaller than the original.

Due to the issues created by sparsity, structured pruning aims for a coarse-grained approach. By removing larger structures of an NN, such as filters or layers, the network maintains its parallel processing structure and avoids sparsity. Filter pruning is used in this work, removing filter channels from convolutional layers (LI et al., 2017). Figure 2.5 (A) shows an example of an RGB image with three channels applied to a convolution with a 3-channel filter, generating a 3-channel feature map, and then the same convolution without the blue filter in Figure 2.5 (B), generating a 2-channel feature map. Applying pruning the blue channel of the filter reduces the size of the kernel from $M \times N \times 3$ to $M \times N \times 2$, and the input's blue channel can be discarded. Removing this filter also decreases the number of channels in the output feature map from $K \times C \times 3$ to $K \times C \times 2$, since each channel corresponds to a filter, leading to roughly a quadratic reduction in memory footprint and computation. For example, reducing the number of filters from a layer by 50% reduces the

Figure 2.4: Execution of time of Dense and pruned (Sparse) networks compared to the Expected pruned times.



Source: Yu et al. (2017).

output by 50%, meaning the next channel's input is also reduced in half. This technique has been used to speedup CNN inference without any hardware modifications.

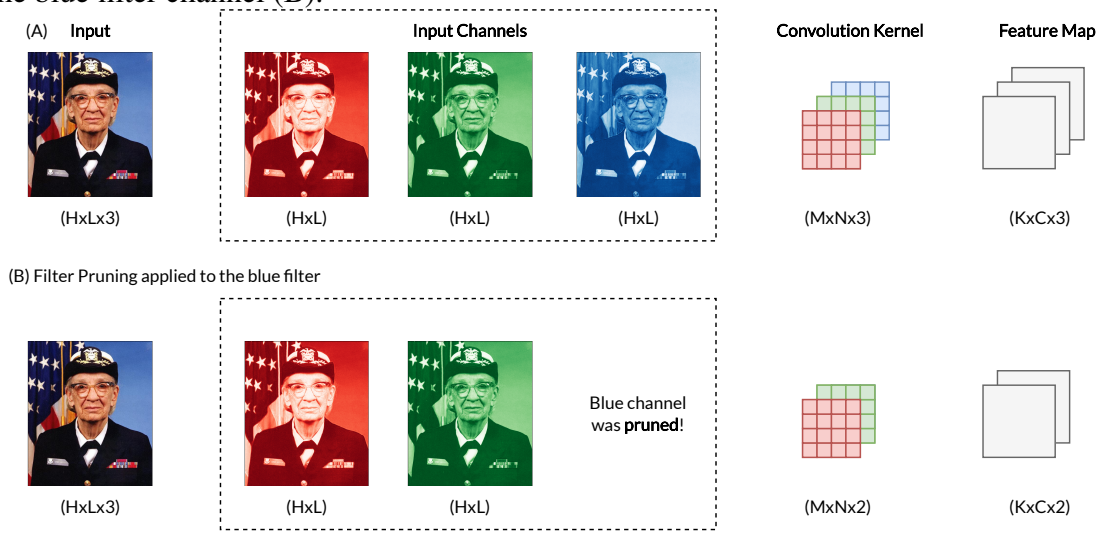
2.4 FPGAs and Hardware Acceleration

Field Programmable Gate Array (FPGA) devices are reconfigurable hardware. Unlike ASICs with static functionality, FPGAs can perform different functions after silicon fabrication (KUON; TESSIER; ROSE, 2007). Their functionality is described via hardware description languages (HDLs) and it is synthesized into a configuration file (bitfile) that maps the function to configurable resources, such as look-up tables (LUTs), digital signal processors (DSPs), block random access memories (BRAMs), and flip-flops (FFs). Figure 2.6 shows an example of a possible architecture of an FPGA device with logic (LUT), memory (FF), multiplier (DSP), and I/O blocks. As a configuration is loaded, all the logic is mapped to these elements and routed accordingly, making FPGAs a very flexible device that can fit many roles, from prototyping to domain-specific task accelerators.

The device's function can be dynamically reconfigured to process different tasks at runtime. This process changes the device's function by loading the bitfile into the device. This process is not instant and takes longer depending on the bitfile's size, the system's memory configuration, and the FPGA device itself.

FPGAs have accelerated machine learning tasks ranging from network intrusion

Figure 2.5: Convolution with a 3-channel filter (A) and a filter-pruned convolution without the blue filter channel (B).



Source: the Author.

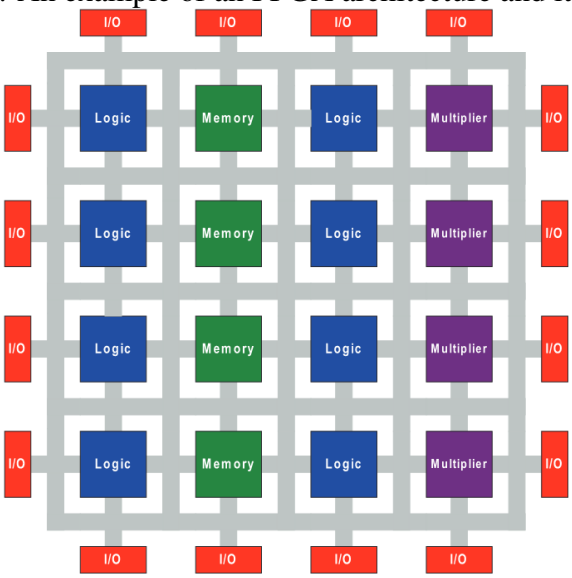
to traffic classification. They provide high throughput and low power consumption and ensure low latency, enabling more computationally intensive algorithms to be employed (ELNAWAWY; SAGAHYROON; SHANABLEH, 2020; TONG; QU; PRASANNA, 2017; QU; PRASANNA, 2015; TONG et al., 2013; LI et al., 2016a). Additionally, FPGAs have been adopted into network infrastructure solutions, such as programmable network interface controllers (LOCKWOOD et al., 2007).

There are two main design categories for accelerators: *single-engine accelerator*, where the same convolutional engine performs all layer's operations, one at a time, and *dataflow accelerators* that use pipeline-based architecture, with different dedicated modules for each CNN layer. This work is based on dataflow accelerators, as they have shown greater performance when compared to their single-engine counterparts due to their specialization and greater exploration of parallelism (BLOTT et al., 2021), as will be discussed next.

2.4.1 FINN

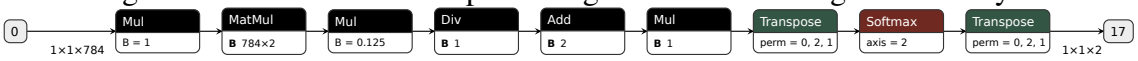
FINN is an open-source framework for generating dataflow hardware accelerators for DNN inference developed by Xilinx (BLOTT et al., 2018; UMUROGLU et al., 2017). All FPGA accelerators in this work employ FINN, as it maps quantized DNN to hardware via modules using High-Level Synthesis (HLS) to configure each layer according to the

Figure 2.6: An example of an FPGA architecture and its resources.



Source: Kuon, Tessier and Rose (2007)

Figure 2.7: An ONNX file representing a DNN with a single hidden layer.



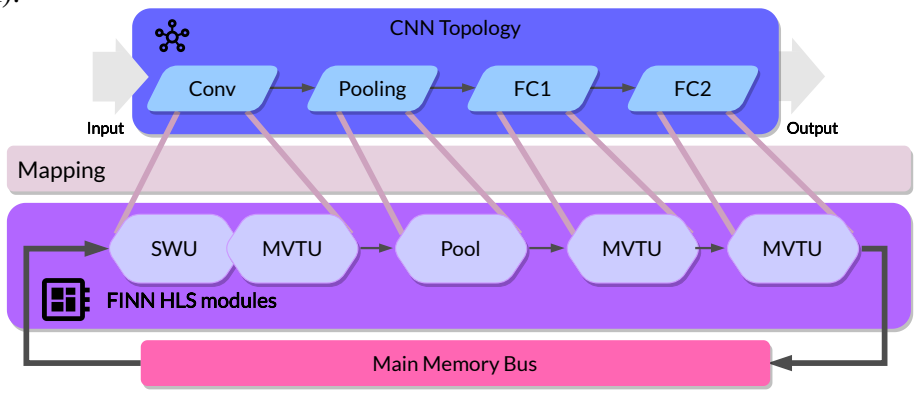
Source: the author.

CNN topology. These accelerators generated by FINN are called dataflow since they are connected in pipeline architecture, where different modules are dedicated to each CNN layer.

FINN uses graph-based descriptions of a DNN's topology in the form of Open Neural Network eXchange format (**ONNX**) files (BAI et al., 2023). Figure 2.7 shows an example of an ONNX file with a DNN with a single fully connected hidden layer. Layers are represented in plain arithmetic operations, such as division (Div), addition (Add), and matrix multiplication (MatMul). This allows for the compatibility of any topology or custom layer with the format, including quantized networks. FINN can infer the different layers (Convolution, max pool, FC, batch normalization) from the graph. A series of transformations are performed in the graph, transforming the description from software to hardware modules. The modules use templates written in C++, which are tailored during runtime to fit the DNN's requirements, topology, and target platform resources.

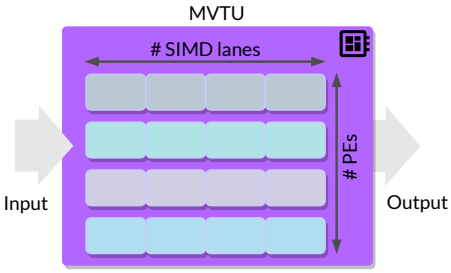
Figure 2.8 shows an example of a CNN mapped to FINN modules. Convolutions can be lowered into a matrix-matrix multiplication (CHELLAPILLA; PURI; SIMARD, 2006). To organize the required inputs for this operation, a convolution layer is transformed into two modules: a Sliding Window Unit (SWU) responsible for organizing the data for convolution into the matrix-matrix format, and a Matrix-Vector-Threshold Unit

Figure 2.8: An example of a CNN mapping from software (top) to a FINN FPGA design (bottom).



Source: the author.

Figure 2.9: An overview of parallelism in FINN's MVTU modules.



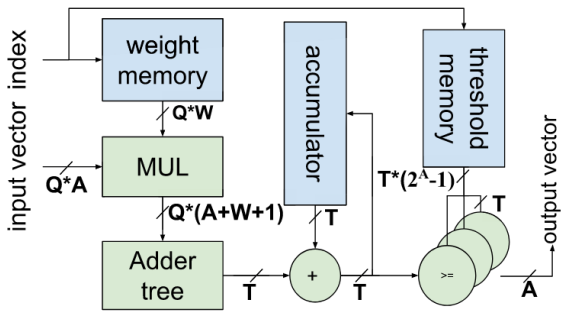
Source: the author.

(MVTU) module performs the quantized operations.

The architecture of an MVTU is shown in Figure 2.9. An MVTU comprises a set of processing elements (PEs) that perform the dot product of a matrix to vector multiplication. Figure 2.10 shows the internal composition of a processing element. **Bold** letters represent the bus' bit-width. Each PE calculates **Q** multiplications in parallel, called **SIMD** value. They are then reduced via an adder tree and a threshold comparison that represents the activation and batch normalization function of the quantized values.

A folding configuration controls the number of **SIMD** lanes and processing elements (**PE**) in an MVTU. Each MVTU in an accelerator has its own set of SIMD and PE,

Figure 2.10: A FINN Processing Element.



Source: Blott et al. (2018).

allowing for a highly customizable design. To find the best folding configuration, FINN has an automatic configuration to maximize throughput and a manual option via folding configuration file for added flexibility and freedom for the user.

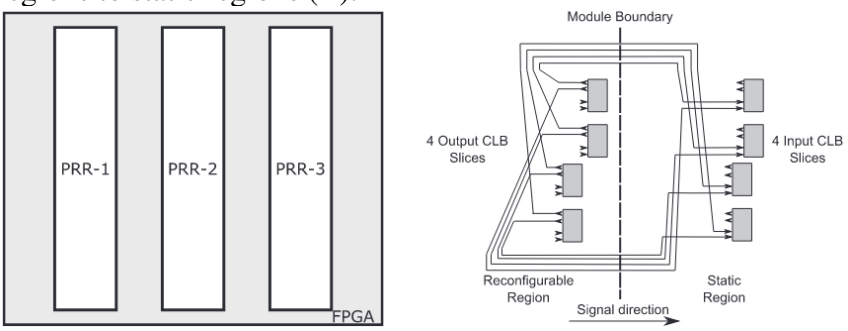
The choice of a folding configuration for each mapped MVTU is essential. It dictates the number of FPGA resources used by the final design and the performance of the generated accelerator. The bigger the values, the larger the design with a higher inference throughput. Two restrictions are imposed on the values for PE and SIMD: the number of output channels in a CONV layer (or neurons in an FC layer) must be a multiple of the layer's PE, and the number of input channels must be a multiple of SIMD lanes. These restrictions ensure that no processing elements or SIMD lanes will be left idle during the processing of a layer, and will fit correctly with the shape of the matrix and vector calculated.

2.5 Dynamic FPGA Reconfigurable Accelerators

Many works explore the reconfigurable nature of FPGAs, proposing solutions to make use of this feature. Two approaches are discussed in this section: dynamic partial (or total) FPGA reconfiguration, which allows for multiple hardware designs to be implemented via time-multiplexed scheduling of bitfile configurations, and overlay hardware architecture, which employs multiple hardware configurations in a single design, switching between them as different tasks are performed.

Using Dynamic Partial Reconfiguration (DPR) allows for blocks of FPGA resources called partially reconfigurable regions (PRR) to be reconfigured while keeping the static regions (SR) with their current configuration (VIPIN; FAHMY, 2019). This enables hardware designs to change some modules during runtime, to optimize a spe-

Figure 2.11: Floorplan of a Virtex-II device (A) and the interface between partial reconfiguration regions to static regions (B).



Source: Vipin and Fahmy (2019).

cific task, with the modules with shared functionality kept, reducing the time and energy used by reconfiguration. Not all FPGA devices include this feature, and the number of PRR blocks available depends on the device architecture. Figure 2.11 (A) shows the three PRRs of a Virtex-II device, and Figure 2.11 (B) shows the communication architecture between PRR and SR. The same interface between PRR and SR must be used in all PRR modules, constraining the I/O in these regions as a single implementation.

3 RELATED WORK

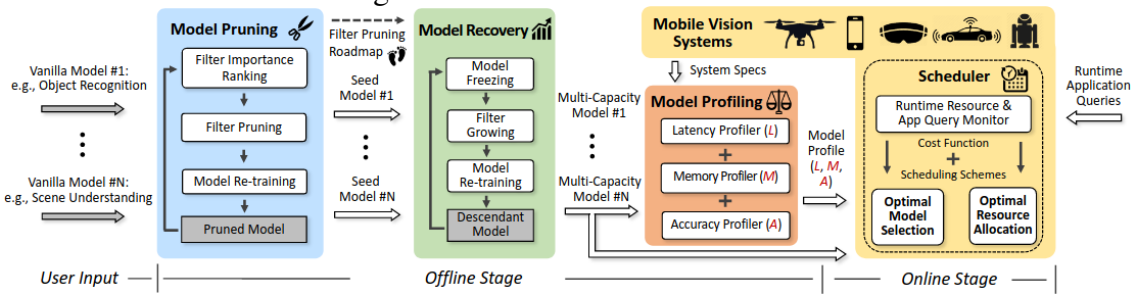
This Chapter presents the State-of-the-Art works closely related to this work. Section 3.1 discusses pruning methods and applications, Section 3.2 explores hardware accelerators for neural networks, Section 3.3 presents works that use dynamic partial reconfiguration on FPGA devices. Finally, Section 3.4 compares this work's proposals to the State-of-the-Art works presented.

3.1 Pruning

The exploration of the resource and accuracy trade-off introduced with pruning was implemented by frameworks such as NestDNN (FANG; ZENG; ZHANG, 2018), ReForm (XU et al., 2019), and DMS (KANG; KIM; PARK, 2019).

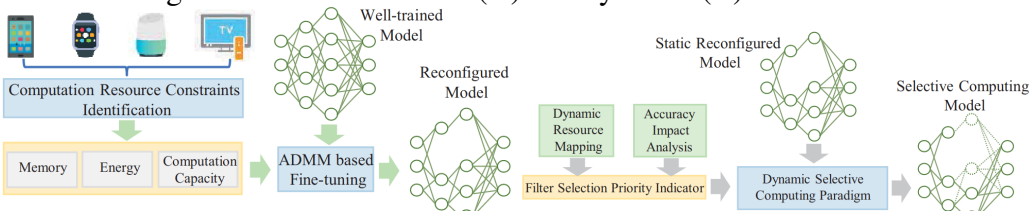
NestDNN focuses on multi-tenant smartphone applications, dynamically scheduling a variant from multi-capacity models generated using pruning based on runtime information. Figure 3.1 shows the NestDNN architecture. Filter prunin is applied to user input "Vanilla Models" (i.e.: DNN models with no pruning), using the *Triple Response Residual* method to select which filters are less relevant in each CONV layer. The number of pruned layers is not set by a percentage of filters to remove, but rather a minimum accuracy value given by the user. From the pruned model, a descendant model is created via *freeze-&-grow* approach, reimplementing previously removed filters. This process creates a series of models with varying sizes and accuracies, that are then profiled. The online stage of the algorithm uses a scheduler to employ the optimal model according to a cost function. Results show that compared to resource-agnostic baseline implementation, NestDNN increases accuracy by up to 4.2%, while doubling the video processing frame

Figure 3.1: NestDNN architecture.



Source: Fang, Zeng and Zhang (2018).

Figure 3.2: ReForm static (A) and dynamic (B) architecture.



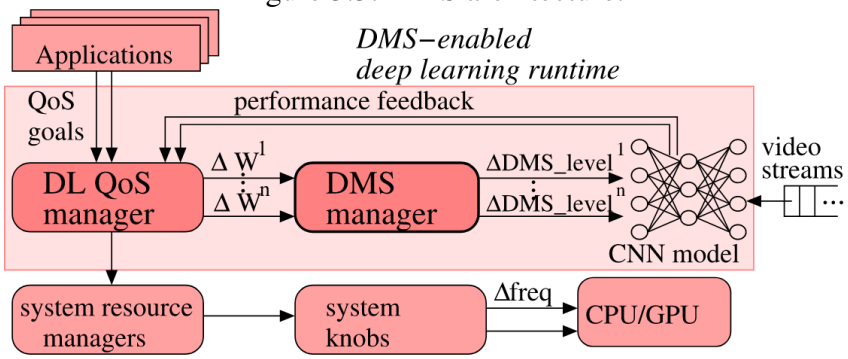
Source: Xu et al. (2019).

rate and reducing energy consumption by $1.7\times$.

ReForm provides a resource-aware inference mechanism in mobile devices. The first step of reform is to identify the constraints of a computing device in terms of memory, energy, and computation capacity. From these metrics, a fine-tuning algorithm is applied to the network using filter pruning to optimize the CNN. ReForm has two modes of operation: a static mode where only the fine-tuning algorithm is applied as seen in Figure 3.2 (A), and a dynamic mode where pruning is used dynamically to a model according to the system's resources, shown in Figure 3.2 (B). The framework is able to reduce costs by up to 18% for workload, 16.23% for latency, 48.63% for memory, and 21.5% for energy.

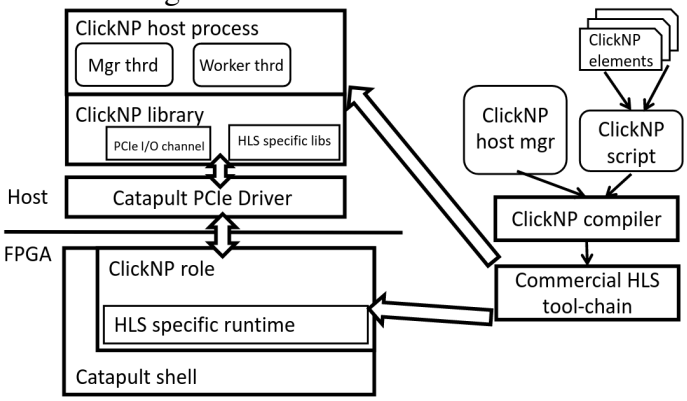
DMS optimizes Quality-of-Service (QoS) by adding and removing filters at runtime based on QoS goals and the number of users changes, in the use-case of video streams. Figure 3.3 shows DMS's architecture. Targeted at deep learning runtimes such as Caffe and TensorRT, the architecture receives application requests for DMS-enable runtimes. During the offline stage, the process rearranges the model of the CNN via filter pruning; instead of removing the filters, the model is maintained complete with reordered filters in order of importance. The application request also has the option to specify a QoS goal, latency, and energy consumption. Their QoS runtime alters the system DVFS and the number of filters used in each convolution to adapt to the incoming inference requests. This framework can efficiently balance the load, especially for unpredictable workloads.

Figure 3.3: DMS architecture.



Source: Kang, Kim and Park (2019).

Figure 3.4: ClickNP architecture.



Source: Li et al. (2016a).

3.2 Neural Network Hardware Accelerators

Many hardware accelerators for network-based machine learning algorithms have been proposed, such as (GROLEAT; ARZEL; VATON, 2014; QU; PRASANNA, 2015; TONG et al., 2013; LI et al., 2016a).

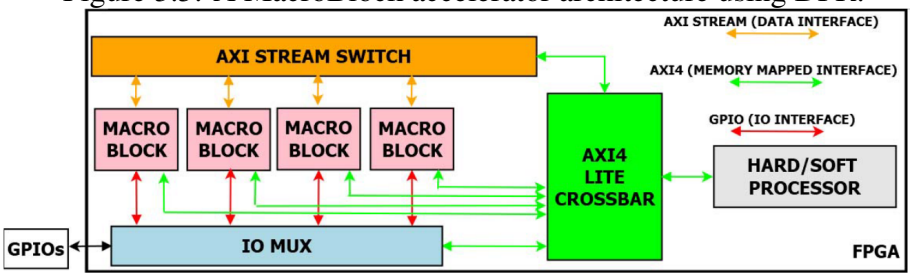
Using support vector machines (SVM) to classify network traffic data, (GROLEAT; ARZEL; VATON, 2014) offer an accelerator architecture for massive parallelism. With some algorithmic changes to make SVM more FPGA friendly and fixed point quantization, the accelerator supports up to 473 GB/s bandwidths running on a Virtex 5 FPGA 187.5 MHz.

Qu and Prasanna (2015) use virtualization to provide hardware acceleration to multiple users for decision tree algorithms. By transforming the tree into a rule-based table, a 2D pipelined architecture is proposed. An engine updates the decision tree for virtualization via deletion, insertion, and modification commands. The accelerator provides up to $5\times$ speedup compared to the other State-of-the-Art tree-based accelerators.

Also focusing on tree classifiers, Tong et al. (2013) presents two accelerator architectures for a decision tree algorithm C4.5 and discretization algorithm Minimum Description Length. The two architectures differ in terms of main memory disposition: one uses distributed on-chip RAM, and the other uses the FPGA's block RAM. The accelerator increases classifications per second from 75-150 million when running on a multicore CPU to 7500 million when using an FPGA. Based on these architectures, a tool is developed to map binary-tree-based classifiers to Verilog code, facilitating accelerator development.

ClickNP (LI et al., 2016a) focuses on facilitating FPGA network function offloading, using a C-like language more familiar to programmers than VHDL; it uses HLS to

Figure 3.5: A MacroBlock accelerator architecture using DPR.



Source: Irmak, Ziener and Alachiotis (2021).

compile the code to HDL language for synthesis. It makes acceleration more flexible and modularized, allowing for joint CPU/FPGA processing. Figure 3.4 shows the architecture proposed. The base for it is Catapult Shell, containing reusable interfaces and logic. To use HLS dynamically, it uses a host process to communicate with the FPGA hardware. This communication allows for parts of the application to be split between FPGA and CPU. Results show it can support up to 200 million packets per second with less than $2\mu s$ latency.

3.3 Runtime Reconfigurable Architectures

The FPGA's capacity to implement multiple functions over time via reconfigurations opens a series of opportunities and challenges to extract the best possible performance on resource-constrained systems.

Exploring CNN accelerators, Irmak, Ziener and Alachiotis (2021) diminishes the throughput and accuracy loss of DPR by proposing a hardware architecture based on HLS MacroBlocks associated with a CNN layer function. MacroBlocks can change their interconnects and functions and even be disabled at runtime. This allows the dynamic insertion, deletion, and updating CNN layers at runtime. The MacroBlocks also employ a pipeline architecture with quantized parameters and the merging of pooling and convolution layers. Figure 3.5 shows the architecture of an accelerator with four MacroBlocks and its connection to a hard/soft processor. As a proof of concept, two LeNet CNN architectures with different output layers and trained on different datasets are implemented. Using a Xilinx Zynq 7020 FPGA running at 100 MHz, the architecture can achieve high accuracy and lower processing time as other State-of-the-Art accelerators, flexibly employing two distinct CNN architectures via DPR.

Approaching the challenge of accelerating large CNN models in resource-constrained FPGA devices, Farhadi, Ghasemi and Yang (2019) uses a novel accelerator architecture

to implement shallow and deep networks in a device. In the context of an MPSoC with an ARM CPU and a Pynq-Z1 FPGA device, a framework that adapts the number of layers during inference, using more or fewer layers depending on the calculated confidence of each layer. Trained models for CIFAR-10, CIFAR-100, and CVHN datasets, the dynamic design can maintain low inference times while maintaining accuracy.

In order to propose an architecture with mid- to high-range devices, Meloni et al. (2016) is adaptable to different convolution layer filter sizes and CNN topologies. In this implementation, CNNs with 5x5 and 3x3 filters are supported, and the internal processing requires input and output buffer adaptation. Moreover, the convolution operation scheduling must be adjusted to the different filter sizes. From the experiments, a Xilinx Zynq XC-Z7045 device achieves up to 120 GMAC/s using 16-bit quantization for 5x5 filters and up to 129 GMAC/s for 3x3 filters using less than 10 W at 150 MHz.

Using the FINN framework (discussed in Section 2.4.1), Seyoum et al. (2021) optimizes performance and resource consumption of DNN inference using DPR to schedule parts of the accelerators at runtime, focusing on finding the optimal decomposition of accelerators minimizing inference time given an area constraint. Using a Zynq-7000 platform, this approach can reduce up to $2\times$ the area, allowing the accelerators to be deployed in area-constrained devices where a static approach would not be possible.

Aiming to lower power consumption on battery-dependent devices, Youssef et al. (2020) uses DPR to explore quantized accelerator models from a 16-bit to a 7-bit design. A VHDL model of the accelerator is employed on a Zynq-7000 board for a CNN trained on the MNIST dataset. The system adjusts the quantization level according to the device's battery level, achieving up to 70% energy reduction at less than 5% accuracy drop.

3.4 Contributions to the State-of-the-Art

Works presented in this chapter offer different methods for optimizing machine learning convolutional neural networks and deploying them dynamically in the context of computer network monitoring. Table 3.1 summarises the comparisons between this study and the related works presented previously. Unlike other works focusing solely on quantization or pruning, this work employs both techniques to offer a bigger design space. This work expands the idea of a static HLS hardware accelerator to a multi-task model, creating greater flexibility than other tailor-made accelerator designs (UMUROGLU et al., 2017; GROLEAT; ARZEL; VATON, 2014; LI et al., 2016a; TONG et al., 2013; QU;

Table 3.1: Comparison w.r.t. the State-of-the-Art.

Work	Quantization	Pruning	HLS	FPGA acceleration	Runtime Adaptability
Umuroglu et al. (2017)	✓		✓	✓	
Groleat, Arzel and Vaton (2014)	✓			✓	
Li et al. (2016a)			✓	✓	
Tong et al. (2013)			✓	✓	
Qu and Prasanna (2015)				✓	✓
Li et al. (2017)		✓			
Fang, Zeng and Zhang (2018)		✓			
Xu et al. (2019)		✓			
Kang, Kim and Park (2019)		✓			
Irmak, Ziener and Alachiotis (2021)	✓		✓	✓	✓
Farhadi, Ghasemi and Yang (2019)			✓	✓	✓
Meloni et al. (2016)	✓			✓	✓
Seyoum et al. (2021)	✓		✓	✓	✓
Youssef et al. (2020)	✓			✓	✓
This work	✓	✓	✓	✓	✓

PRASANNA, 2015). Moreover, instead of employing dynamically reconfigurable designs via dynamic partial reconfiguration, this work proposes a hardware virtual layer that aims to reduce the number of FPGA reconfigurations for workloads that require multiple tasks performed over time, as will be discussed in the following chapter.

4 FRAMEWORK AND HARDWARE SOLUTIONS

This chapter explores the implementation of **Anya**: a framework to dynamically reconfigure the FPGA according to the number of incoming inferences requests, presented in Section 4.1, **Tara**: a framework that allows for multiple different CNNs to be executed using the same FPGA bitfiles, detailed in Section 4.2, and **Spike**: a combination of the two optimizations, presented in Section 4.3.

4.1 Anya: Reconfiguration Framework

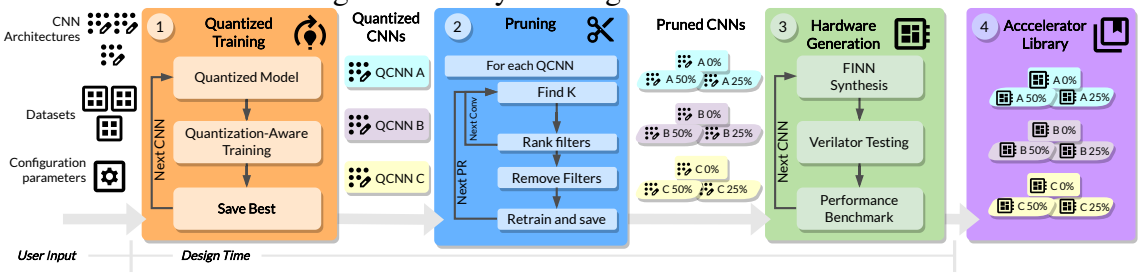
The Anya framework is the base for the optimizations proposed in this work. Anya is based on a hardware system composed of an FPGA-enabled smart network interface card (SmartNIC) via FINN (as detailed in Section 2.4.1) as its main use case. SmartNICs can process packets without host support to implement various tasks, from intrusion detection to traffic classification, the latter used in this work. The different classification tasks are requested one at a time, receiving a variable amount of network traffic, creating a dynamic environment where adaptability is required. Each task requires a specially trained CNN, which can be accelerated by custom hardware acceleration solutions, enabling the demanded adaptability and efficiency in this context. Anya exploits the throughput-accuracy trade-off through different versions of custom accelerators generated at design time. The set of accelerators creates a design space explored by a runtime algorithm that selects the best design according to the requested classification task (i.e., specific CNN) and network traffic condition (i.e., number of packets to be classified).

Anya uses a two-step solution. First, the offline design-time phase generates the library of accelerators for each task. Next, the runtime phase continually monitors the network conditions to select the most appropriate accelerator. Thus, this runtime search may reconfigure the FPGA whenever necessary (changing either task or pruning rate). The steps are explained in more detail next.

4.1.1 Design Time

Figure 4.1 shows an overview of Anya's design time and will guide the explanation in this section. Design time's main goal is the generation of a library of pruned

Figure 4.1: Anya's design-time overview.



Source: the author.

accelerators with varied accuracy-performance profiles for each classification task. The three steps are ① **Quantized Training**, ② **Pruning**, and ③ **Hardware Generation**. These are performed for each task, considering **user input**: each CNN architecture and their respective dataset and the configuration settings, specifying training hyperparameters, hardware's folding configurations, pruning rates, and system-specific characteristics, such as FPGA boards. In the example in Figure 4.1, three distinct CNNs (i.e.: classification tasks) with three datasets are input by the user, named A, B, and C.

① **Quantized Training:** The first step towards designing an FPGA accelerator is through quantization. Initially, the user must input each classification task CNN topology, alongside their dataset and training hyperparameters. The quantization phase is paramount, as it will greatly impact the amount of resources required to implement hardware arithmetic units and the memory space required to store the trained parameters.

Figure 4.1 ① shows each step of the training process. Currently adapting an existing unquantized pytorch or tensorflow model requires user input and is not automatic, as the CNN model must be adapted into a different library: Brevitas. Brevitas is the the quantization-aware library based on PyTorch (PAPPALARDO, 2021) selected to implement quantized versions of the CNNs, as it is part of Xilinx's FINN framework for dataflow accelerator generation. Brevitas uses extended classes from pytortch, making the transition to quantized models a simple change (e.g.: a Conv1d layer becomes a QuantConv1d), and most of the effort comes from parameters to be adjusted by the user, such as bitwidth and number representation (integer, fixed-point, etc).

With an adapted brevitas CNN model, quantization-aware training does not require any further changes, and follows the same flow as any other pytorch model. This training flow already considers the reduction in arithmetic accuracy during the backpropagation instead of post-training quantization, making it more robust and allowing for more aggressive quantization.

Following the number of training epochs as the user requested, Anya always chooses to save the parameters that achieve the highest test accuracy after each epoch. This ensures that overtraining issues will not arise, and grants confidence to the user to train each CNN for as long as their time budget allows.

Each quantized CNN is saved, as shown in Figure 4.1 ①, named as QCNNs (A, B, and C). Once trained, the parameters are saved into pytortch format for pruning, and as an ONNX file to be transformed into an accelerator using the FINN workflow.

② **Pruning:** Figure 4.1 ② shows the pruning steps that are applied to all trained QNNs from the previous step. Filter pruning generates a set of CNN models (as shown previously in Figure 2.5). FINN imposes two restrictions in terms of its folding configuration (i.e.: parallelism constraints, as discussed in Section 2.4.1): the number of output channels in a CONV layer (or neurons in an FC layer) must be a multiple of the layer's PE, and the number of input channels must be a multiple of SIMD lanes. These conditions ensure the full utilization of the hardware during execution. Based on a user input folding configuration and the original CNN, the method adjusts the pruning so that the generated hardware will adapt to FINN's requirements.

For each convolutional layer, given a pruning rate $0 < pr < 1$, the method attempts to remove $k = Ch_{original}\ pr$ channels, following the restrictions in Equation 4.1 and 4.2.

$$(Ch^{i}_{out} - k) \bmod PE_i = 0 \tag{4.1}$$

$$(Ch^{i}_{out} - k) \bmod SIMD_{i+1} = 0 \tag{4.2}$$

Where PE_i and $SIMD_{i+1}$ are the number of PE and SIMD lanes of the current layer i and following layer $i+1$, and Ch^{i}_{out} is the number of output channels in the current layer (notice that $Ch^{i}_{out} = Ch^{i+1}_{in}$). In the case of a convolutional layer followed by an FC layer, the SIMD condition changes to Equation 4.3.

$$(Ch^{i}_{out} - k)FM \bmod SIMD_{i+1} = 0 \tag{4.3}$$

Where FM is the FC layer's input flattened feature map size.

If the value of k does not meet these conditions, it is decremented by one. The process repeats until either the conditions are met or $k = 0$, the latter meaning that the convolution layer cannot be pruned under this pruning rate and these folding restrictions.

With the number of pruned channels defined, it is necessary to rank the channels in order of importance by a defined criterion, to evaluate the ones with the least impact in the convolution. In this work, the pruning criterion selected is filter pruning, as presented by (LI et al., 2017). The method calculates for each filter in a trained convolutional layer the sum of its absolute weight values (i.e., the ℓ_1-norm). The k filters with the lowest sum are pruned. Once all convolutional layers are pruned, the new CNN model is retrained, its test accuracy is saved, and the model is exported as an ONNX file.

Once the pruning step is complete, a set of new CNNs for each task will form our design space. In the example in Figure 4.1 three pruning rates were considered: 0, 25, and 50%, generating a total of nine CNNs for all three tasks.

③ **Hardware Generation:** Given the pruned and unpruned ONNX files generated in the previous step, FINN is used to synthesize each custom Dataflow accelerator. Figure 4.1 ③ shows the FINN synthesis using the user's folding configuration. In this step, the ONNX files (shown in Figure 2.7 are mapped into hardware modules in HLS code. The HLS code is synthesized to the user's selected FPGA part and tested for correctness via verilator. This uses a dataset input and output from the software version to ensure the same results are found in the hardware accelerator.

FINN reports for inference throughput, and power are gathered from synthesis, with clock accuracy measurements, as well as the TOP-1 accuracy results from each pruned task training phase, and stored alongside their bitfile.

From this, it is possible to evaluate the inference performance of the various design points and their accuracy-performance trade-off. With all CNNs synthesized, Anya has a set of accelerators for each classification task input by the user to be used at runtime, as shown in Figure 4.1 ④ with a set of nine distinct accelerators.

4.1.2 Runtime step

During runtime, all traffic passing through the SmartNIC is considered for classification. This leads to lost inferences in moments where the number of packets in the network surpasses the processing capabilities of the system, undermining the application.

Algorithm 1 shows the pseudocode for Anya's runtime execution. Given the hardware library generated in the previous step (Figure 4.1 ④), Anya overcomes this high demand by continually adapting the SmartNIC inference according to the current traffic flow. First, the current number of inferences to be processed and the classification task

Algorithm 1 Pseudocode for Anya's runtime accelerator selection

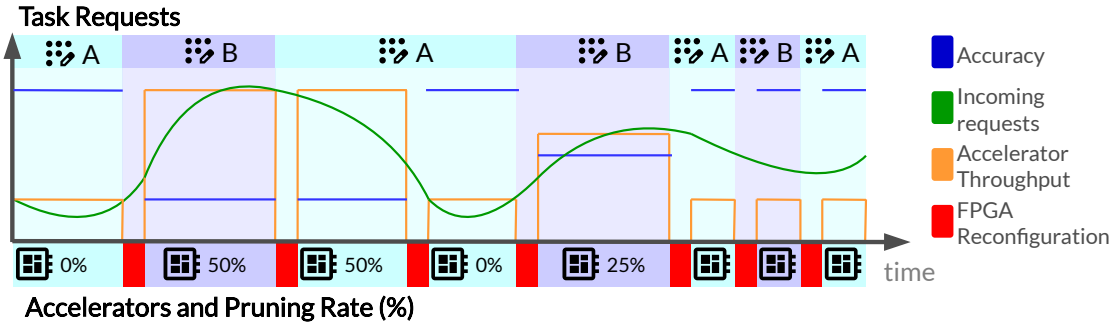
Require: $0 < minDeltaValue < 1$

1: **function** ANYA_RUNTIME($AccLibrary, minDeltaValue$)
2: $LastFps \leftarrow 0$
3: $LastTask \leftarrow 0$
4: **loop**
5: $Fps \leftarrow GetRequests()$ ▷ Evaluates the number of incoming packets
6: ▷ Calculates the workload change
7: $FpsDelta \leftarrow |Fps - LastFps|$
8: ▷ Calculates minimum workload change required for reconfiguration
9: $MinDelta \leftarrow minDeltaValue \times LastFps$
10: $CurrentTask \leftarrow GetTask()$ ▷ Receives the current classification task
▷ Reconfigure on task change or workload variation
11: **if** $LastTask \neq CurrrentTask \vee FpsDelta > MinDelta$ **then**
12: $Models \leftarrow GetClassModels(AccLibrary, CurrentTask)$
13: ▷ Applies Equation 4.4
14: $ConfigId \leftarrow CalculateBestQoEModel(AccLibrary, Fps)$
15: $RecongigureFPGA(ConfigId)$
16: **end if**
17: $LastFps \leftarrow Fps$
18: $LastTask \leftarrow CurrentTask$
19: **end loop**
20: **end function**

is read from the SmartNIC, and the difference from the previous value is calculated on line 9; Two conditions dictate the search of a different accelerator: (i) when the requested task differs from the current one, or (ii) when the incoming number of inference requests increases or decreases by more than x% of the current workload. This value can be fine-tuned by the user. As tested for the current use-case DNN and workload scenario, it was empirically set as 25% and presented in Algorithm 1 as $minDeltaValue$. In these cases, the FPGA is reconfigured by the framework, with the intent of optimizing the quality of experience (QoE) of the system. These conditions are verified in line 11.

If an FPGA reconfiguration is required, Anya first filters the library to select only accelerators that perform the current task (line 12). Then the algorithm for accelerator selection takes as input the current number of incoming inferences (Fps) and the benchmarked information in the $AccLibrary$ during design time. The criteria for selection is calculated based on the current workload and the achievable quality of experience of each accelerator in the library for that task, following the formula in Equation 4.4, where W_{model} indicates the maximum throughput of the accelerator, W_{input} the current workload and α the model's accuracy. If the accelerator is capable of processing more frames than the current workload, then it considers the maximum percentage of processed frames of

Figure 4.2: Example of Anya's runtime.



Source: the author.

1 (100%), represented by $min(\cdot)$.

$$QoE_{max} = min(\frac{W_{model}}{W_{input}}, 1)\alpha \tag{4.4}$$

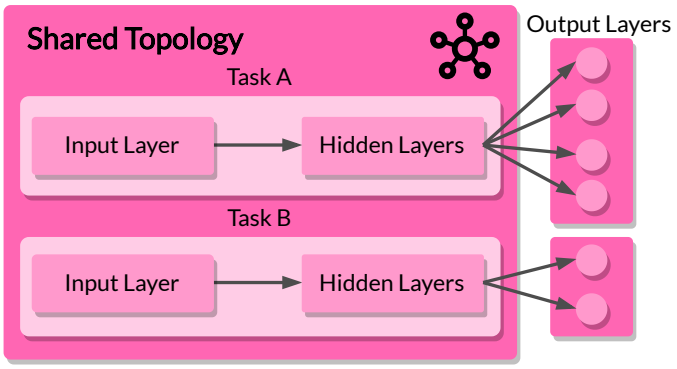
Equation 4.4 is evaluated on each accelerator in the set for the requested task. Then, the one with the highest value is selected for FPGA configuration as the $ConfigId$. The FPGA is finally reconfigured on line 15. Even if no reconfiguration occurred, lines 17-18 update the last FPS and task currently being processed for the next iteration of the loop.

To better illustrate Anya in action, Figure 4.2 shows a graph with tasks A and B requested over time (x-axis). In this example scenario, the number of incoming inference requests varies over time (green curve), and the FPGA accelerator configured and their pruning rate is displayed at the bottom, with a red rectangle representing the reconfiguration time overhead, and the accelerator's throughput and accuracy are represented by the orange and blue curves.

Following Figure 4.2, the example starts with task A and a low number of incoming tasks, allowing Anya to select an unpruned accelerator (0% pruning rate). On the request for task B, the incoming requests increase greatly, so Anya changes the FPGA accelerator and selects a pruning rate of 50%. Notice that as the accelerator throughput increases, the accuracy achieved is diminished. As the request switches back to task A, Anya keeps the pruning rate of 50%, but as the requests fall sharply, Anya selects a more suitable unpruned version of the accelerator with 0% pruning and much greater accuracy. Following this, Anya switches to a 25% pruning rate for task B.

It's important to note that FPGA reconfiguration is not instantaneous, and this process will cause performance degradation if performed too frequently. This can be viewed in the example in Figure 4.2, in the last three task requests. The reconfiguration

Figure 4.3: Example of two classification tasks with shared topology and different output layers.



Source: the author.

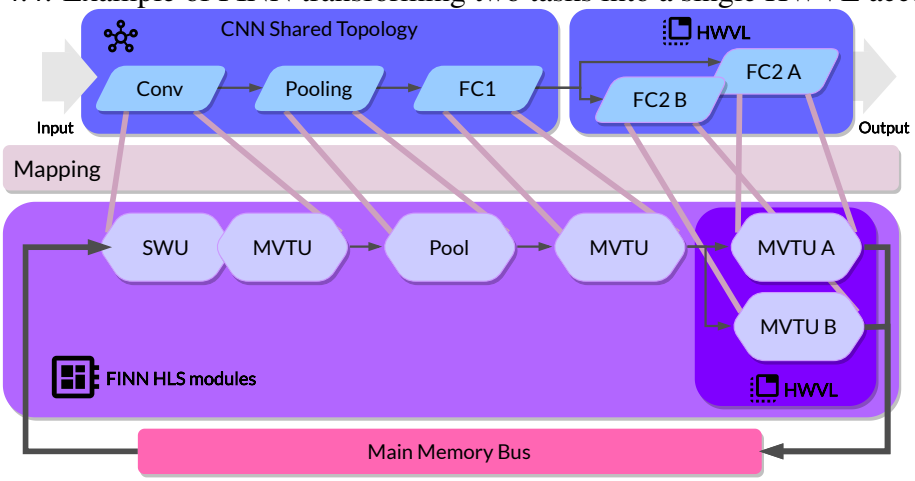
time in this case takes up almost half the time the accelerators are deployed and running. This constant switching can lead to great degradation in performance and is not something Anya can handle currently.

4.2 Tara: Hardware Virtual Layers

Frequently switching FPGA models comes at a cost: time and energy spent loading the configuration file to the board. This quickly adds up and the overhead cost of changing bitfiles can completely negate the speedup granted by the accelerator. This undermines the reconfigurability potential of the device, as both time and energy efficiency are crucial for hardware acceleration. To explore tailor-made dataflow accelerators capable of performing multiple classification tasks without reconfiguration, this section presents Tara: a solution using hardware virtual layers (HWVL).

In a system where multiple classification tasks are deployed using the same base deep neural network architecture, many of the same hardware modules are identical between designs. Precisely, some examples of networks that fit these criteria can be employed for network intrusion detection (WANG et al., 2017b) or for encrypted protocol classification (WANG et al., 2017a). Tara is a solution for deploying multiple deep neural network accelerators using a single design, removing the need for FPGA reconfiguration when switching from one classifier to another for CNNs that share topology up to the last layer. An example of CNNs that fit this criterion, Figure 4.3 shows two classification tasks with shared topology. They have the same input size and the same hidden layers, with different trained parameters. Their output layer differs in the number of classes according to each task. The shared layers can be converted into the same dataflow accelerator modules.

Figure 4.4: Example of FINN transforming two tasks into a single HWVL accelerator.

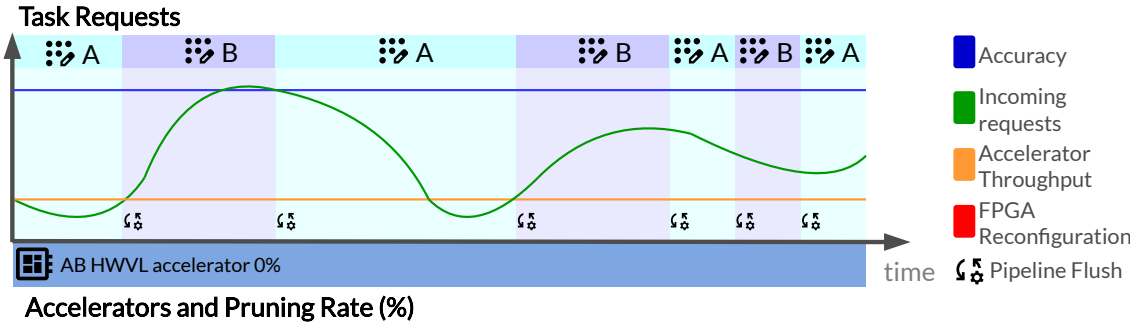


Source: the author.

The idea of a hardware virtual layer is to implement a single FPGA bitfile design capable of accelerating the different tasks without extra FPGA reconfiguration. The virtual layer is employed via hardware replication of FINN HLS modules (as detailed in Section 2.4.1). The proposed accelerators follow FINN's transformations, as shown previously in Figure 2.8; the last layer of a CNN is converted into an HLS module (MVTU) tailored to that layer. Figure 4.4 shows an example of two CNN classifiers transformed into a single accelerator. The shared topology between the tasks is implemented using the same modules, while the last layers of the classifiers (inside the HWVL rectangles) have dedicated modules. Instead of implementing a single MVTU for the last layer, the hardware uses multiple MVTUs, one for each classification task. All of these MVTUs are connected to the previous layer's output, but only one is active for processing based on the current task.

Anya has trouble when too many different task requests are made in a short period, due to the reconfiguration overhead time. In the case of Tara, to switch tasks the hardware first finishes the inferences currently being processed, and only when they are finished the new task can start processing inferences again. This takes the same time as the entire pipeline latency, which is the time it takes to process a single inference. It is called a *pipeline flush* in this work, following the terminology used in pipelined CPU cores. This means that when using Tara, the overhead time it takes to change the current classification task is reduced from several hundred milliseconds to a single clock cycle, as the accelerator pipeline can output an inference per cycle. This is the main selling point of Tara: quickly switching classification tasks at will at almost no time overhead cost at the cost of more FPGA resource usage by the replicated modules.

Figure 4.5: Example of Tara's runtime.



Source: the author.

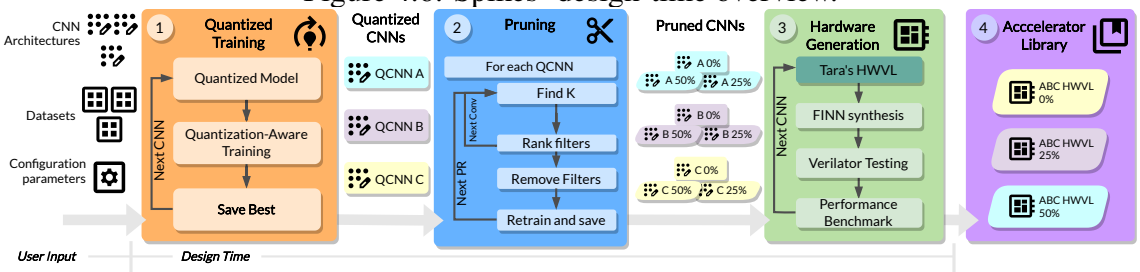
To better illustrate this point, Figure 4.5 shows the same task requests and incoming requests rates as Anya's Figure 4.2, so a direct comparison between the approaches can be made. Since Tara can implement both tasks, no time is spent in FPGA reconfiguration, and each time a new task is requested a simple pipeline flush is performed, indicated by the cog and arrow symbol. Since the time overhead for a flush is so small, it is not indicated in the accelerator throughput curve. In this case, the last three tasks that proved problematic to Anya become trivial for Tara. However, since Tara uses the original CNN without pruning, the accelerator throughput is not enough in most cases. This points to a need to combine optimization efforts, as will be discussed next.

4.3 Spike: a combined effort

To benefit from Anya's dynamic pruning and Tara's quick task switching, Spike combines both approaches. Figure 4.6 shows an overview of Spike, from user input, quantized training, and pruning steps following Anya, to the new and modified hardware generation step and accelerator library.

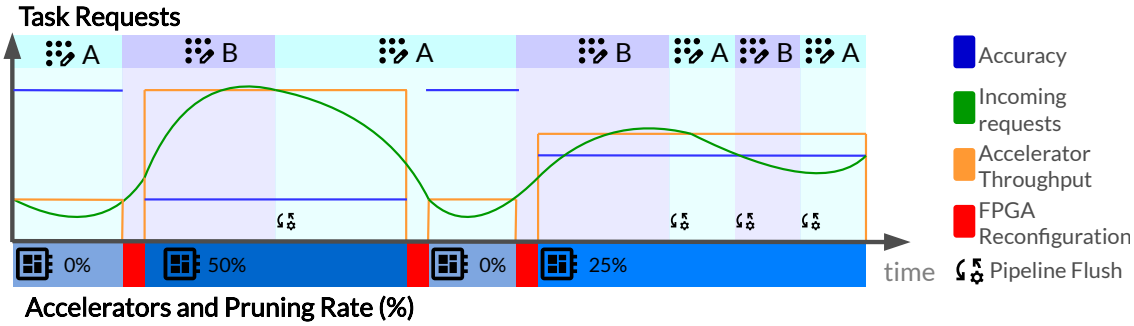
Since steps ① and ② have already been detailed in Section 5.2.2, we start the

Figure 4.6: Spikes' design-time overview.



Source: the author.

Figure 4.7: Example of Spikes's runtime.



Source: the author.

explanation from a set of quantized and pruned CNNs in this section.

③ **Hardware Generation**. In this step, the pruned CNNs must go through Tara's HWVL integration. So, for each pruning rate, the tasks are combined in their shared topology and a split is made for their HWVL implementation by creating a custom ONNX file. Once Tara's implementations are complete, FINN synthesis is performed followed by testing and benchmarking. In this case, each bitfile will be associated with a set of different accuracies, respective to their tasks. In this example, a single bitfile will be able to perform tasks A, B, and C. This will form the accelerator library.

④ **Accelerator Library**. The number of bitfiles generated is dictated by the set of pruning rates input by the user. In the example, three pruning rates (0, 25, and 50%) were used, generating only three bitfiles. Compared to the example in Figure 4.1, Spike reduces the number of bitfiles from 9 to 3, while maintaining the functional capabilities and throughput improvements from pruning, at the cost of more FPGA resources and static power consumption from the additional MVTU modules required to implement HWVL.

To better illustrate the benefits of Spike, Figure 4.7 displays the same task requests and incoming requests rates as the previous examples in Figures 4.2 and 4.5. Task A starts with a 0% pruning rate accelerator, followed by a sharp increase of incoming requests and a task switch to task B. Here, an FPGA reconfiguration is required to change pruning rates, using a 50% accelerator. No reconfiguration is required on the task switch back to A, only a pipeline flush. In the same task A, incoming requests drop quickly and Spike reconfigures the FPGA to a 0% pruning rate accelerator again. Finally, the incoming requests rise once again, leading to a reconfiguration for task B. Here many task switches are requested but only flushes are performed. Notice that throughout the example the number of incoming requests is consistently below the accelerator throughput, and the

time spent in FPGA reconfiguration is much shorter than in the previous two examples, displaying the effectiveness of Spike's optimizations.

5 DISCUSSION

In this chapter, the methodology and experiments are presented in Section 5.1, followed by a comprehensive discussion of the results in Section 5.2. The results stem from the accelerator library generated in Subsection 5.2.1, and further insights are provided on the Anya framework runtime in Subsection 5.2.2, Tara's hardware virtual layers in Subsection 5.2.3, and the collaborative efforts of Spike in Subsection 5.2.4.

5.1 Methodology

This section presents the applications, tools, hardware, and metrics used for experiments presented in this chapter.

5.1.1 Dataset

For the evaluation of the proposed Anya framework and virtual layer hardware, the encrypted VPN traffic classification problem is selected. The ISCX VPN-nonVPN traffic dataset (DRAPER-GIL et al., 2016) contains raw traffic data labeled by protocol use (VPN or not VPN) and the application type that generated the traffic. This work divides the raw traffic into *four* different representations based on: session or flow. A flow is represented by the same source IP and destination IP, meaning it only considers packets flowing from one IP to the other, while a session considers packets going both ways. Both are characterized by a packet's 5-tuple data (source IP, source port, destination IP, destination port, and transport-level protocol). From these two representations, they are further separated by which protocol layers are used for the dataset samples: using *only* the seventh layer of the TCP/IP model (L7) or all layers (All). From this, the work presents four distinctive representations of the same dataset named: Flow+L7, Flow+All, Sesssion+L7, Session+All. Based on the results presented in Wang et al. (2017a), Session+All is used in this work, as it shows the best accuracy results. Furthermore, this representation of the raw traffic more closely reflects how a SmartNIC would receive packets for classification in a real setting. The samples in the dataset are 784 bytes long and labeled into 12 classes, differentiating between regular (non-VPN) and protocol (VPN) encrypted traffic and by their type: email, chat, streaming, file transfer, VoIP, or P2P.

5.1.2 CNN applications

From the selected dataset, the work in Wang et al. (2017a) presents four possible classification problems can be trained from the Session+All dataset, each representing a different classification task:

- **2 classes**: classify packets between VPN and non VPN;
- **6 classes VPN**: classify VPN packets by traffic type;
- **6 classes nonVPN**: classify non-VPN packets by traffic type;
- **12 classes**: classifies both traffic type and protocol (VPN or non-VPN).

From these four tasks, four respective CNN architectures are proposed. To replicate these CNNs into an FPGA accelerator used as the tasks for experiments in Anya's hardware library during the design time step, and Tara's HWVL design counterparts, modifications were required to fit into FINN's dataflow design.

- Instead of 32-bit floating point inputs, weights, bias, and activations, these parameters were quantized, using 2-bit for inputs and 4-bit for weights, bias, and activations. This modification reduces the models' maximum achieved accuracy, as will be discussed in Section 5.2.1.
- Batch normalization layers are added, as FINN's quantization technique can exploit them, allowing for the normalization calculation to be absorbed by the hardware's quantization layer (UMUROGLU; JAHRE, 2018), as explained in more detail in Section 2.4.1.
- The size of the max pool kernels is increased from 3 to 4.

Table 5.1 details the four different CNN models, comparing our adapted versions with the original State-of-the-Art implementation by Wang et al. (2017a) (differences are highlighted in **bold**).

The main difference between each task regarding CNN architecture is their output layers, varying from 2 to 12 neurons, corresponding to the task's classification classes. Once trained, they also present different parameter values (weights and biases) but maintain a shared topology up to the final FC layer.

Table 5.1: CNN topology of the evaluated models, comparing our adapted CNN architecture (top) to the original CNNs (bottom). The final layer differs in the number of outputs for each task. All CONV and FC layers implement bias.

Adapted CNNs					
Layer	**Input**	**Kernel size**	**Stride**	**Padding**	**Output**
1D Convolution + **BatchNorm** + ReLU	784*1	25	1	same	784*32
1D Max Pool	784*32	**4**	**4**	same	**196*32**
1D Convolution + **BatchNorm** + ReLU	**196*32**	25	1	same	**196*64**
1D Max Pool	196*64	**4**	**4**	same	**49*64**
Fully Connected + **BatchNorm** + ReLU	**49*64**	-	-	none	1024
Fully Connected + **BatchNorm** + SoftMax	1024	-	-	none	2/6/12

State-of-the-Art CNNs					
Layer	**Input**	**Kernel size**	**Stride**	**Padding**	**Output**
1D Convolution + ReLU	784*1	25	1	same	784*32
1D Max Pool	784*32	**3**	**3**	same	**262*32**
1D Convolution + ReLU	**262*32**	25	1	same	**262*64**
1D Max Pool	**262*64**	**3**	**3**	same	**88*64**
Fully Connected + ReLU	**88*64**	-	-	none	1024
Fully Connected + SoftMax	1024	-	-	none	2/6/12

Source: Partially adapted from (WANG et al., 2017a).

5.1.3 Quantized Training

Each model is trained using the PyTorch-based library Brevitas (PAPPALARDO, 2021). It is an open-source tool by AMD/Xilinx used for quantization-aware training and is part of the FINN framework (discussed in Section 2.4.1). Each of the four tasks was trained for 500 epochs with a learning rate of 0.01, using stochastic gradient descent with a minibatch size of 50. Pruned CNNs are retrained using the same hyperparameters for an additional 150 epochs. The TOP-1 test accuracy results are reported.

5.1.4 FPGA accelerators

All accelerators used for the experiments were generated using Xilinx's FINN Framework (BLOTT et al., 2018) and synthesized using Xilinx Vivado (VIVADO..., 2023) for resource usage and power information, and Verilator is used for RTL simulations of performance (VERIPOOL, 2023).

The selected target platform is an Alveo U200 Data Center Acceleration card at 100 MHz, which is commonly used as a smartNIC and network accelerator card. It comprises 1303k LUT, 2607k FF, 6840 DSP, and 2016 BRAM units.

The average FPGA reconfiguration time is variable, depending on factors such

as FPGA part, disk I/O, PCIe latency, bitmap size, and many other overheads (Xilinx, 2020; XILINX, 2019; MOODY, 2021). From the previous references, the FPGA reconfiguration is set at an optimist 300ms for all experiments, lower than the estimate of the reference, in detriment to our proposal. The optimist value favors the baseline of our experiments, as real-world values would likely be larger, meaning that if experiments show performance improvements with a short reconfiguration time, even better results could be achieved with a larger reconfiguration time.

As discussed in Section 2.4.1, FINN provides control of the design's parallelism via the parameters PE (number of processing units) and SIMD (the number of SIMD lanes) of each MVTU module via a folding configuration. These parameters are fixed in the designs to evaluate the pruned accelerator's processing capabilities fairly. They are constrained equally across the different pruning rating design points. This approach reduces the number of changing variables in the experiment, as the objective is not to explore the hardware's parallelism but rather the speedup granted by pruning.

This chapter presents the methodology and experiments in Section 5.1, followed by a discussion of the results found in Section 5.2, from the accelerator library generated in Subsection 5.2.1, the Anya framework runtime in Subsection 5.2.2, Tara's hardware virtual layers in Subsection 5.2.3, and the Spike's combined effort in Subsection 5.2.4. Each accelerator is composed of four MVTUs, corresponding to the CNN's CONV and FC layers. Table 5.2 shows each layer's fixed PE and SIMD constraints, based on FINN's automatic folding configuration for these hardware implementations. These constraints ensure that the same number of parallel operators are used for the different accelerator models, ensuring a fair comparison between the different pruned accelerators. Since the four tasks share the same topology for the first three layers, they use the same values, while the last layer of PE constraint follows the number of classes from the different tasks. In the case of HWVL accelerators that implement all four classification tasks in a single design, these constraints are also valid.

To generate pruned accelerators for Anya and Tara, the Dataflow-Aware pruning method is applied with rates from 5% to 80% at 5% steps. Given the relation between the pruning rate and the number of PE and SIMD lanes discussed in Section 2.3.2, six unique accelerators can be generated with pruning rates of 15%, 25%, 40%, 50%, 65%, and 75%, alongside an unpruned version.

Table 5.2: Folding configuration for each MVTU module of the generated FINN accelerators.

Layer	PE	SIMD
0	4	5
1	8	4
2	32	8
3	2/6/12	32

Source: the author

5.1.5 Evaluation Scenarios and metrics

To evaluate Anya, Tara, and Spyke, four scenarios are proposed that vary in terms of workload and the frequency new tasks are requested:

- **Stable Workload (S)**: varying the number of incoming packets by 12.5% every 8 seconds;
- **Variable Workload (V)**: varying the number of incoming packets by 75% every 2 seconds;
- **Low Switching rate (L)**: a different task is requested every 15 seconds;
- **High Switching rate (H)**: tasks change every 2 seconds.

The combination of these characteristics forms four scenarios, called **SH, SL, VH,** and **VL**, covering the different network conditions. Each scenario represents very particular network conditions: a stable workload with few input changes, representing the moments when the network is constantly requested at a very predictable rate, without any huge drops or spikes in the workload, making the processing requirements almost constant. A variable workload aims to test scenarios with very unpredictable workloads when the number of incoming packets can vary greatly each second, making room for more optimizations as the number of inferences required is constantly changing. For the stable scenarios, the dynamic nature of our proposal can focus on selecting the best accelerators only a few times, while in the variable scenarios, a more aggressive change in accelerators might be required due to constantly changing processing requirements. Similarly, the difference between low and high switching rate scenarios focuses on stressing two opposite ends in the multitasking optimizations. Experiments execute each scenario for 360 seconds.

To compare the performance of our solutions, the unmodified FINN hardware models with no pruning are used as **baseline** (Table 5.1), and each time a new task is requested the FPGA is reconfigured, similar to the example shown in Figure 1.1.

The metrics used to evaluate the experiments are Quality of Experience (QoE), performance, and energy per inference. QoE is defined in Equation 5.1, where P_{proc} is the number of processed packets, P_{total} is the total packets that were requested for classification, and α represents the classification accuracy.

$$QoE = \frac{P_{proc}}{P_{total}}\alpha \tag{5.1}$$

5.2 Results

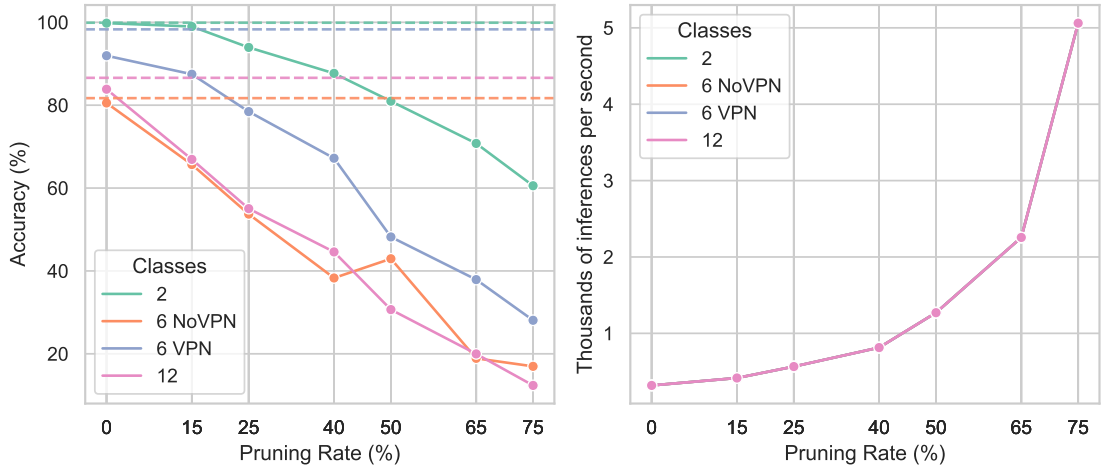
This section presents the results from the experiments detailed in the previous section. First, the design space of the pruned accelerator library using different pruning rates is explored in subsection 5.2.1, then Anya is evaluated in subsection 5.2.2, followed by Tara in subsection 5.2.3, and Spike in subsection 5.2.4. Finally, an overall comparison and overview of results is discussed in Section 5.2.5

5.2.1 Static design space exploration

Initially, the design space created by the library of pruned accelerators generated by Anya's design step (presented in Section 4.1.1) is analyzed. Here, the results present a library *without* HWVL, as the impact of Spyke using HWVL on throughput will be discussed at the end of this Section. Figure 5.1 (A) shows the impact pruning has on each classifier, with the x-axis showing the percentage of channels pruned and the y-axis showing the achieved accuracy for post-training evaluation. At the *leftmost* side of the graph, the pruning rate is **zero**, with the *dashed* lines representing the accuracies of the state-of-the-art implementation shown in (WANG et al., 2017a). It is clear that even with quantization, the unpruned classifiers achieve accuracy levels close to the original models, with accuracy drops of 0.20, 1.17, 6.37, and 3.30% for 2, 6 VPN, 6 nonVPN, and 12 classes tasks, respectively. Some accuracy reduction is expected when using quantization due to the reduced precision of calculations. Since all of our models use 2-bit inputs and 4-bit inputs, the comparison to the original models that use 32-bit floating point arithmetic (WANG et al., 2017a), the accuracy loss is justifiable for a representation reduction of $8\times$ for parameters, and $16\times$ for inputs.

The impact of pruning differs for each task. Noticeably, 6 nonVPN and 12 classes

Figure 5.1: Accuracy vs. pruning rate (A) and inferences per second vs. pruning rate (B) for each accelerator model.



Source: the author

tasks have a steep accuracy reduction, even with only 15% pruning, whereas 2 and 6 VPN tasks maintain a smoother decrease in accuracy for up to 40% pruning. This is because 6 nonVPN and 12 classes original models already achieve lower accuracy than 2 and 6 VPN, requiring more parameters to achieve great accuracy. This makes the pruning of parameters affect accuracy more harshly. The inferences per second are similar across tasks as they overlap in the graph, and they increase exponentially with the pruning rate, as shown in Figure 5.1 (B). On the 2 classes task, Anya can increase the inferences per second from 318, with no pruning, to 565, at 25% pruning, with a 6.58% accuracy drop. On the 12 classes task, a more significant accuracy drop (28.82%) is observed for the same performance increase. This leads to a diversity of optimization opportunities produced by the trade-off of accuracy and number of inferences in each task, which the dynamic part Anya will exploit.

In the case of Spike's HWVL pruned library, throughput and accuracy remain the same as presented above. This is because the virtual layers don't change the datapath of the accelerator. The only fundamental change is the addition of multiple final modules that can be switched during runtime, which have no impact on throughput or accuracy.

5.2.2 Anya: runtime evalutation

Using the library of accelerators evaluated previously in Subsection 5.2.1, Table 5.3 shows the runtime results for the Anya framework compared to the baseline on the

four scenarios described in Methodology's Subsection 5.1.5.

Average QoE baseline refers to the difference between baseline QoE and the compared framework, averaged from the entire runtime of each experimental scenario. Thus, comparing the baseline with itself results in zero. Considering this metric, Anya achieves from 5.08 to 7.91% increase in QoE compared to the baseline. Scenarios with high task switching rates (SH and VH) present the most improvement. This is due to two main factors: First: the number of FPGA reconfigurations in each scenario is closely associated with the number of optimization opportunities Anya can perform by changing the current employed pruning rate. Second: if the baseline implementation has a large number of FPGA configurations, Anya's increase in reconfigurations will lead to less overhead relative to the baseline implementation. This is clear when comparing VH and VL. In VH, the increase in the number of reconfigurations from baseline and Anya is 34, while for VL that number rises to 107. The overhead clearly reduces the gains in QoE for VL, leading to the least improvements. On the opposite end is SH, where Anya increases the number of reconfigurations by only 16, leading to the best QoE in this set of experiments. The amortization and the flexibility to reconfigure the FPGA 151 times leads to Anya selecting the best possible accelerator for each moment, with many opportunities to fine-tune the current accelerator to best fit a moment-to-moment input change.

In terms of increase in processed frames, VH and VL provide the best conditions for Anya to excel in its reconfiguration strategy, leading to up to $1.40\times$ increase in processed frames. This is because these scenarios best utilize the throughput gain of pruned accelerators. They are significantly faster than their unpruned counterparts used as the baseline, as shown previously in Figure 5.1, and volatile scenarios provide sufficient incoming frames to make the best use of this optimization. This is contrasted by the SH and SL scenarios, which still have a great performance, with 1.26 and $1.24\times$ respectively. Since the stable scenarios don't change greatly, Anya has fewer opportunities to employ higher pruning rate accelerators, leading to lower gains.

The metric energy per inference refers to is tied directly to the throughput of pro-

Table 5.3: Comparison of Baseline and Anya.

Scenario	**SH**		**SL**		**VH**		**VL**	
Algorithm	**Baseline**	**Anya**	**Baseline**	**Anya**	**Baseline**	**Anya**	**Baseline**	**Anya**
Average QoE to Baseline	0.00	7.91	0.00	6.30	0.00	7.38	0.00	5.08
Increase in QoE to Baseline	1.00	1.13	1.00	1.09	1.00	1.11	1.00	1.08
Processed Frames to Baseline	1.00	1.26	1.00	1.24	1.00	1.40	1.00	1.36
Energy per Inference (mJ)	13.88	10.84	11.73	9.30	13.88	9.86	11.73	8.97
# FPGA Reconfigurations	135	151	18	23	135	169	18	125

Source: the author.

cessed inferences using the least amount of power. The most reduction comes from VH, followed by VL, as they also show the most increase in processed frames, with 1.4 and $1.3\times$ less energy per inference when compared to the baseline implementation. Both SH and SL show similar decreases in 1.26 and $1.28\times$. Overall, the gains in energy efficiency show that even though the framework uses an algorithm based on maximizing QoE, it also leads to energy savings due to the higher number of processed frames enabled by pruning.

5.2.3 Tara: runtime evaluation

Table 5.4 shows the results comparing Tara and the baseline implementation for the four scenarios of experiments.

Tara's main strength is the ease of switching accelerator configurations and this is directly reflected in the results's average QoE to baseline. In scenarios where tasks must be switched frequently (SH, VH), Tara increases QoE in 1.13 and $1.07\times$. In the case of VH, the increase is diminished by the fact that variable workloads have higher inference processing requirements, and Tara is bound to unpruned accelerators. This is also reflected in scenarios SL and VL, where the long periods between task switches leave no room for optimization. Overall, there are many benefits in terms of QoE for Tara, given its potential is fully utilized.

The gains in QoE are a direct reflection of the increase in the number of processed frames. Here the time overhead of reconfiguration is fully displayed: the benefits of using Tara's HWVL lead to an increase in $1.13\times$ processed frames for both SH and VH. These are the scenarios where the number of reconfigurations required drops from 135 to only the initial FPGA configuration. The number of pipeline flushes reflects this, and even being performed 179 times in these scenarios, it does not interfere with performance in any way.

The extra hardware required to implement Tara's HWVL comes at a cost in terms

Table 5.4: Comparison of baseline and Tara.

Scenario	**SH**		**SL**		**VH**		**VL**	
Algorithm	**Baseline**	**Tara**	**Baseline**	**Tara**	**Baseline**	**Tara**	**Baseline**	**Tara**
Average QoE to Baseline	0.00	8.03	0.00	1.04	0.00	4.31	0.00	0.38
Increase in QoE to Baseline	1.00	1.13	1.00	1.02	1.00	1.07	1.00	1.01
Processed Frames to Baseline	1.00	1.13	1.00	1.01	1.00	1.13	1.00	1.01
Energy per Inference (mJ)	13.88	12.63	11.73	12.63	13.88	12.63	11.73	12.63
# FPGA Reconfigurations	135	1	18	1	135	1	18	1
# Pipeline Flushes	0	179	0	23	0	179	0	23

Source: the author.

of energy. Since more FPGA resources are required to implement the same functions, it results in higher static and dynamic power compared to the baseline.

The method shows a marginal increase in energy per inference in the SL and VL scenarios, due to the increase in static power consumption for very little use of its ease of task switching. Indeed, in this scenario, Tara only performs 23 pipeline flushes. The greater use of FPGA resources and the energy per inference, when the Tara can be fully utilized, is great, and it leads to minor increases in energy otherwise.

5.2.4 Spyke: runtime evaluation

Table 5.5 shows the results comparing Spyke to the baseline implementation. Spyke combines the benefits of Anya's framework for dynamic pruning and reconfiguration and the flexibility of Tara's task switching via HWVL, and this is reflected in the results.

When considering average QoE to baseline, Spyke has a significant impact, especially on the SH scenario. This is expected, as this is the scenario that could benefit the most from both Anya and Tara's optimizations, further boosting Spyke's combined optimization effort. With an increase in QoE of $1.22\times$ on average, a stable scenario with a high task switching frequency allows Spyke to change pruning rates frequently while also adapting to new tasks quickly. This is reflected in the number of FPGA reconfigurations, dropping from 135 at baseline to only 52. The variable and high-frequency switching scenario (VH) also has a great improvement with spike, but suffers from a less drastic decrease in reconfigurations, with only 16 less than the baseline, while performing 61 pipeline flushes. This means that the framework can effectively change pruning rates and find optimization opportunities, but the overhead from reconfiguration reduces the QoE improvements to $1.13\times$. The gains in SL and VL scenarios are still significant. Indeed, while Tara proved not too useful in these cases, Spyke can improve QoE by 1.10 and

Table 5.5: Comparison of baseline and Spyke.

Scenario	**SH**		**SL**		**VH**		**VL**	
Algorithm	**Baseline**	**Spyke**	**Baseline**	**Spyke**	**Baseline**	**Spyke**	**Baseline**	**Spyke**
Average QoE to Baseline	0.00	13.81	0.00	6.79	0.00	8.41	0.00	5.25
Increase in QoE to Baseline	1.00	1.22	1.00	1.10	1.00	1.13	1.00	1.08
Processed Frames to Baseline	1.00	1.37	1.00	1.24	1.00	1.46	1.00	1.37
Energy per Inference (mJ)	13.88	10.28	11.73	10.09	13.88	10.11	11.73	9.67
# FPGA Reconfigurations	135	52	18	15	135	119	18	114
# Pipeline Flushes	0	128	0	10	0	61	0	11

Source: the author.

$1.08\times$ respectively. Especially in the scenario SL, where the baseline uses 18 reconfigurations, Spyke reduces the reconfigurations by 3, while allowing HWVL to be used 10 times for task switching with no overhead.

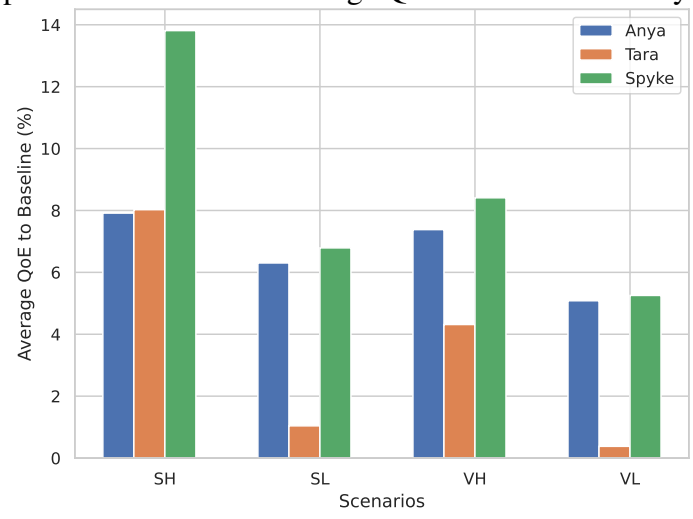
In terms of increase in processed frames, SH and VH achieve a 1.37 and $1.46\times$ increase respectively. The reduction in reconfiguration overhead granted by Spyke's HWVL allows for more time to be spent processing inferences, leading to even granted throughput. VL also shows an increase in $1.37\times$, meaning that although the QoE increase is not as significant, Spyke can fully utilize the throughput increase of pruned accelerators, compensating for the number of FPGA reconfigurations. Lastly, SL maintains a significant increase in processed frames of $1.24\times$.

The impact of Spyke on energy proves significant. Considering the metric of energy per inference, even with the additional power required by Tara's HWVL, it is possible to achieve up to $1.37\times$ reduction in energy for the VH scenario. This is once again tightly related to the increase in processed frames, as discussed above. Even in scenario SL, where Spyke has the least impact, it can reduce energy consumption by $1.16\times$, proving that even in scenarios where Tara's HWVL would not be ideal, Spyke can leverage its adaptability and achieve positive results.

5.2.5 Comparisons and Oportunities

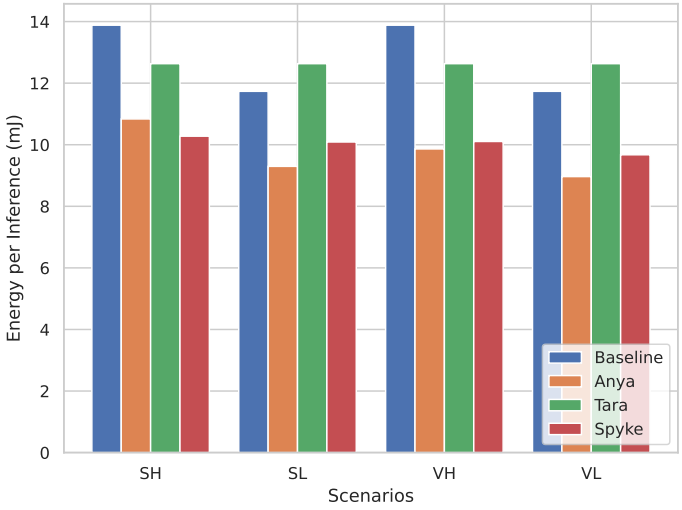
This section presents an overall comparison of the three presented optimization frameworks and how they compare to each other for the experimental scenarios presented.

Figure 5.2: Comparison of increase in average QoE to baseline for Anya, Tara, and Spyke.



Source: the author

Figure 5.3: Comparision of energy per inference for Baseline, Anya, Tara, and Spyke.



Source: the author

Figure 5.2 presents a side-by-side comparison of the Average QoE to baseline for Anya, Tara, and Spyke. Here we can see that Spyke always performs better than the separate use of the other two frameworks, granting better QoE than the baseline. It's most clear its performance improvements on the SH scenarios, where Anya and Tara both had similar performance improvement of around 8%, with Spyke reaching close to 14%. It's also clear that Spyke sees less improvements compared to Anya in scenarios where Tara doesn't perform as well: SL and VL, but it still sees some minor gains. Overall, it's clear that Spyke is a net benefit for the average QoE compared to using either Anya or Tara separately.

In terms of energy gains for each framework, Figure 5.3 shows a comparison of energy per inference compared to baseline. Here, we find that Anya has the lowest energy consumption in scenarios SL and VL, followed by Spyke. Indeed, as we can see, Tara uses a substantially higher energy per inference compared to the other two in all scenarios. This is the trade-off of using HWVL, as they use more FPGA resources and thus require more power, leading to greater energy consumption. However, Spyke is still able to overcome this in the scenario SH, where it finds the lowest energy consumption of all three. Overall, in terms of energy, Anya finds the best results, closely followed by Spyke.

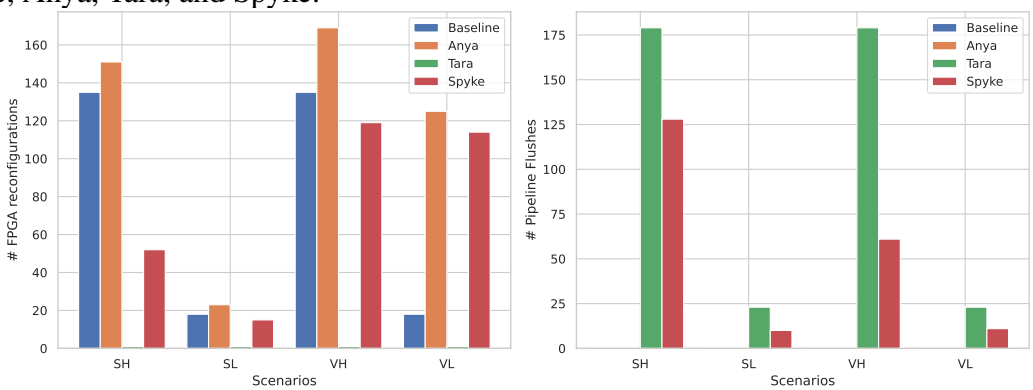
Figure 5.4 shows the comparison of FPGA reconfigurations and pipeline flushes for each framework in the different scenarios. Here we can see the impact of the reconfiguration time overhead and the number of optimization opportunities each framework took. A pipeline flush means that Tara or Spyke made use of the fast task switching of HWVL, saving the need for an FPGA reconfiguration. This is most useful for SH and

VH, where task switching is constant.

As for the case of FPGA reconfigurations, we can see the number of baseline reconfigurations. This is the exact number of task switching each scenario requests. From this, we can glean how many other reconfigurations Anya and Spyke used a different pruning rate accelerator. This is most clear in the VL and VH scenarios, where the volatile change in incoming requests leads to greater opportunities for pruning and fine-tuning.

Overall, our experiments show a diverse set of network conditions and task requests, benefiting the use of each framework. In terms of QoE, Spyke is always able to provide benefits, while in terms of energy, Anya can deliver the best results in scenarios with less frequent task requests. We can see that Spyke is still able to benefit from both optimizations, always outperforming our baseline.

Figure 5.4: Comparision of reconfigurations (left) and pipeline flushes (right) for Baseline, Anya, Tara, and Spyke.



Source: the author

6 CONCLUSION

In this MSc dissertation, we investigated FPGA reconfiguration techniques for machine learning accelerators using quantization, pruning, and hardware virtual layers to employ multiple packet classification tasks from network interface cards with variable bandwidth. The number of inferences required for the framework varies over time, allowing for the dynamic employment of accelerator models and designs, and optimizations to allow for more flexible task switching between executions.

In exploring hardware design and CNN reduction optimizations, we proposed three optimization frameworks: Anya, Tara, and Spyke. Anya employs a library of pruned FPGA accelerators dynamically switching as different tasks and throughputs are required. Tara employs hardware virtual layer architectures on accelerators, allowing for fast task switching. Spyke uses a combination of Anya and Tara, using a library of HWVL accelerators dynamically.

In our use case of four VPN classification tasks, the pruned State-of-the-Art CNNs show different accuracy drop-offs in terms of pruning rate, some tasks are more suited for pruning, with small decreases, while others present a high accuracy drop even with a low pruning rate. The acceleration granted by the pruning optimization is similar in all tasks, showing an almost exponential increase in throughput with higher pruning rates. The static performance of the library points to many optimization opportunities for the different tasks, as their accuracy vs. throughput trade-offs are diverse.

When considering the Anya in action in various scenarios, the experiments show that compared to our baseline implementation using only accelerators with no pruning, Anya is always able to achieve an increase in Quality of Experience (QoE) and energy reduction, with an increase of up to $1.13\times$, and reducing the energy per inference by up to $1.40\times$ granted by the use of dynamically reconfigured pruned accelerators. Tara shows the most improvements where fast task switches are required, with up to $1.13\times$ increase in QoE. In scenarios where task switching is not as frequent, still offers some minor improvements in QoE and energy.

We find the best results of combining both approaches in Spyke, with an increase in $1.22\times$ in QoE and $1.37\times$ more processed frames with a reduction of up to $1.35\times$ energy. The reduction of reconfigurations of Tara's HWVL and the addition of dynamic pruning configurations is the optimal way to lead with a multi-task inference FPGA acceleration.

Overall, the use of techniques to improve the use of FPGAs and explore their reconfiguration capabilities can lead to energy reductions and increased performance.

6.1 Future Work

Spyke can be further developed aiming at its two main aspects: the framework and the hardware. The use of reconfiguration could be further explored in dynamic partial reconfiguration, making use of individually pruned layers, instead of entire networks. This would allow for finer-grained pruning, a larger design space, and faster reconfiguration times. This would require more network models to be trained for each variation but could lead to more optimization opportunities.

Similarly, the same concept could be applied to Tara's hardware virtual layers, where each layer of the network could be turned virtual, removing the reconfiguration requirements completely. As seen from the presented results, reconfiguration is a great overhead cost in terms of performance and energy, and a completely virtual architecture that is still tailor-made to a set of network models could lead to great flexibility and performance gains.

Tara could be further improved with a module design that fits the idea of reconfigurable pruning more closely. This work uses the FINN architecture as a basis, but there are many opportunities for exploring other processing element designs that could be dynamically configured during runtime.

Finally, for Spyke an accelerator model algorithm that considers the particularities of the HWVL could lead to an even greater exploration of Tara's HWVL architecture, as the algorithm currently does not consider reconfiguration times for FPGA model selection.

6.2 Publications

As a result of the work developed in the course of the Master's program, the following publications have been made:

- VICENZI, J. C. et al. Adaptive Inference on Reconfigurable SmartNICs for Traffic Classification. In: BAROLLI, L. (Ed.). Advanced Information Networking and Applications. Cham: Springer International Publishing, 2023. (Lecture Notes in

Networks and Systems), p. 137–148. ISBN 978-3-031-28451-9

- VICENZI, J. C. et al. Dynamic Offloading Decisions for Improved Performance and Energy Efficiency in Heterogeneous IoT-Edge-Cloud Continuum. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2023.
- VICENZI, J. C. et al. TRIPP: Transparent Resource Provisioning for Multi-Tenant CPU-GPU based Cloud Environments. In: 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC). [S.l.: s.n.], 2021. p. 1–8. ISSN: 2324-7894.
- KNORST, T. et al. On the benefits of Collaborative Thread Throttling and HLS-Versioning in CPU-FPGA Environments. In: 2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI). [S.l.: s.n.], 2022. p. 1–6.

REFERENCES

ACETO, G. et al. Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. **IEEE Trans. Netw. Serv. Manag.**, v. 16, n. 2, p. 445–458, 2019.

BAI, J. et al. **ONNX: Open Neural Network Exchange**. [S.l.]: GitHub, 2023. <https://github.com/onnx/onnx>.

BENGIO, Y.; LÉONARD, N.; COURVILLE, A. C. Estimating or propagating gradients through stochastic neurons for conditional computation. **CoRR**, abs/1308.3432, 2013. Available from Internet: <http://arxiv.org/abs/1308.3432>.

BLOTT, M. et al. Evaluation of optimized cnns on heterogeneous accelerators using a novel benchmarking approach. **IEEE Trans. on Comp.**, v. 70, n. 10, p. 1654–1669, 2021.

BLOTT, M. et al. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, USA, v. 11, n. 3, p. 1–23, 2018.

BOUTABA, R. et al. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. **J. Internet Serv. Appl.**, v. 9, n. 1, p. 16:1–16:99, 2018.

CAO, Z. et al. A Survey on Encrypted Traffic Classification. In: BATTEN, L. et al. (Ed.). **Applications and Techniques in Information Security**. Berlin, Heidelberg: Springer, 2014. (Communications in Computer and Information Science), p. 73–81. ISBN 978-3-662-45670-5.

CHELLAPILLA, K.; PURI, S.; SIMARD, P. High Performance Convolutional Neural Networks for Document Processing. Suvisoft, 2006.

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

CISCO. QoS: Classification Configuration Guide. 2023.

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

DRAPER-GIL, G. et al. Characterization of Encrypted and VPN Traffic using Time-related Features:. In: **Proceedings of the 2nd International Conference on Information Systems Security and Privacy**. Rome, Italy: SCITEPRESS - Science and and Technology Publications, 2016. p. 407–414. ISBN 978-989-758-167-0.

ELNAWAWY, M.; SAGAHYROON, A.; SHANABLEH, T. Fpga-based network traffic classification using machine learning. **IEEE Access**, v. 8, p. 175637–175650, 2020.

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.

FARHADI, M.; GHASEMI, M.; YANG, Y. **A Novel Design of Adaptive and Hierarchical Convolutional Neural Networks using Partial Reconfiguration on FPGA**. arXiv, 2019. ArXiv:1909.05653 [cs]. Available from Internet: <http://arxiv.org/abs/1909.05653>.

GHOLAMI, A. et al. A survey of quantization methods for efficient neural network inference. n. arXiv:2103.13630, 2021. Available from Internet: <http://arxiv.org/abs/2103.13630>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

GROLEAT, T.; ARZEL, M.; VATON, S. Stretching the edges of svm traffic classification with fpga acceleration. **IEEE Trans. on network and service management**, IEEE, v. 11, n. 3, p. 278–291, 2014.

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

IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2019 (Revision of IEEE 754-2008)**, p. 1–84, 2019.

IRMAK, H.; ZIENER, D.; ALACHIOTIS, N. Increasing Flexibility of FPGA-based CNN Accelerators with Dynamic Partial Reconfiguration. In: **2021 31st International Conference on Field-Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2021. p. 306–311. ISSN: 1946-1488.

JACOB, B. et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. **CoRR**, abs/1712.05877, 2017. Available from Internet: <http://arxiv.org/abs/1712.05877>.

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.

KRISHNAMOORTHI, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. **CoRR**, abs/1806.08342, 2018. Available from Internet: <http://arxiv.org/abs/1806.08342>.

KUON, I.; TESSIER, R.; ROSE, J. FPGA Architecture: Survey and Challenges. **Foundations and Trends® in Electronic Design Automation**, v. 2, n. 2, p. 135–253, 2007. ISSN 1551-3939, 1551-3947. Available from Internet: <http://www.nowpublishers.com/article/Details/EDA-005>.

LI, B. et al. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In: **SIGCOMM**. [S.l.]: ACM, 2016. p. 1–14.

LI, H. et al. Pruning filters for efficient convnets. **CoRR**, abs/1608.08710, 2016. Available from Internet: <http://arxiv.org/abs/1608.08710>.

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

LOCKWOOD, J. W. et al. Netfpga-an open platform for gigabit-rate network switching and routing. In: **MSE**. [S.l.]: IEEE Computer Society, 2007. p. 160–161.

MELONI, P. et al. A high-efficiency runtime reconfigurable IP for CNN acceleration on a mid-range all-programmable SoC. In: **2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.: s.n.], 2016. p. 1–8.

MOODY, K. P. **FPGA-Accelerated Digital Signal Processing for UAV Traffic Control Radar**. Thesis (PhD) — Brigham Young University, 2021.

NURVITADHI, E. et al. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In: **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. Monterey California USA: ACM, 2017. p. 5–14. ISBN 978-1-4503-4354-1. Available from Internet: <https://dl.acm.org/doi/10.1145/3020078.3021740>.

PAPPALARDO, A. **Xilinx/brevitas**. Zenodo, 2021. Available from Internet: <https://doi.org/10.5281/zenodo.3333552>.

QU, Y. R.; PRASANNA, V. K. Enabling high throughput and virtualization for traffic classification on FPGA. In: **FCCM**. [S.l.]: IEEE Computer Society, 2015. p. 44–51.

RETSINAS, G. et al. Weight pruning via adaptive sparsity loss. **CoRR**, abs/2006.02768, 2020. Available from Internet: <https://arxiv.org/abs/2006.02768>.

SEYOUM, B. et al. Spatio-Temporal Optimization of Deep Neural Networks for Reconfigurable FPGA SoCs. **IEEE Transactions on Computers**, v. 70, n. 11, p. 1988–2000, nov. 2021. ISSN 1557-9956. Conference Name: IEEE Transactions on Computers.

TONG, D.; QU, Y. R.; PRASANNA, V. K. Accelerating decision tree based traffic classification on FPGA and multicore platforms. **IEEE Trans. Parallel Distributed Syst.**, v. 28, n. 11, p. 3046–3059, 2017.

TONG, D. et al. High throughput and programmable online trafficclassifier on fpga. In: **FPGA**. [S.l.: s.n.], 2013. p. 255–264.

UMUROGLU, Y. et al. Finn: A framework for fast, scalable binarized neural network inference. In: **FPGA**. [S.l.]: ACM, 2017. p. 65–74.

UMUROGLU, Y.; JAHRE, M. Streamlined Deployment for Quantized Neural Networks. **arXiv:1709.04060 [cs]**, may 2018. ArXiv: 1709.04060.

VANHOUCKE, V.; SENIOR, A.; MAO, M. Z. Improving the speed of neural networks on cpus. In: **Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011**. [S.l.: s.n.], 2011.

VERIPOOL. **Verilator**. 2023. Accessed: 2023-24-04. Available from Internet: <https://www.veripool.org/verilator/>.

VINAYAKUMAR, R.; SOMAN, K. P.; POORNACHANDRAN, P. Applying convolutional neural network for network intrusion detection. In: **2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)**. [S.l.: s.n.], 2017. p. 1222–1228.

VIPIN, K.; FAHMY, S. A. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. **ACM Computing Surveys**, v. 51, n. 4, p. 1–39, jul. 2019. ISSN 0360-0300, 1557-7341. Available from Internet: <https://dl.acm.org/doi/10.1145/3193827>.

VIVADO ML Overview. 2023. Available from Internet: <https://www.xilinx.com/products/design-tools/vivado.html>.

WANG, W. et al. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In: **2017 IEEE International Conference on Intelligence and Security Informatics (ISI)**. Beijing, China: IEEE, 2017. p. 43–48. ISBN 978-1-5090-6727-5.

WANG, W. et al. Malware traffic classification using convolutional neural network for representation learning. In: **2017 International Conference on Information Networking (ICOIN)**. [S.l.: s.n.], 2017. p. 712–717.

XILINX. **Get Moving with Alveo**. 2019. Accessed: 2022-01-08. Available from Internet: <https://www.xilinx.com/developer/articles/example-0-loading-an-alveo-image.html>.

Xilinx. **PetaLinux Tools Documentation**. 2020. Accessed: 2022-01-08. Available from Internet: <http://xilinx.eetrend.com/files/2020-06/wen_zhang_/100049850-99702-petalinuxgongjuwendangcankaozhinan.pdf>.

XU, Z. et al. Reform: Static and dynamic resource-aware DNN reconfiguration framework for mobile device. In: **DAC**. [S.l.: s.n.], 2019. p. 1–6.

YOUSSEF, E. et al. Energy Adaptive Convolution Neural Network Using Dynamic Partial Reconfiguration. In: **2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)**. [S.l.: s.n.], 2020. p. 325–328. ISSN: 1558-3899.

YU, J. et al. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In: **Proceedings of the 44th Annual International Symposium on Computer Architecture**. Toronto ON Canada: ACM, 2017. p. 548–560. ISBN 978-1-4503-4892-8. Available from Internet: <https://dl.acm.org/doi/10.1145/3079856.3080215>.

ZHANG, A. et al. Dive into deep learning. **arXiv preprint arXiv:2106.11342**, 2021.

ZHANG, W. et al. A framework for resource-aware online traffic classification using CNN. In: **CFI**. [S.l.]: ACM, 2019. p. 5:1–5:6.