TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TALITA TOBIAS CARNEIRO

DISTRIBUTION OF LOW LATENCY MACHINE LEARNING ALGORITHM

Master of Science thesis

Examiner: Prof. Timo Hämäläinen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 29th August 2018

ABSTRACT

TALITA TOBIAS CARNEIRO: Distribution of Low Latency Machine Learning Algorithm
Tampere University of Technology
Master of Science thesis, 71 pages
December 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Timo Hämäläinen
Keywords: Neural Networks, Inference Accelerator, FPGA, Machine Learning, Ultra-Low Latency, Mobile Networks

Mobile networks are evolving towards centralization and cloudification while bringing computing power to the edge, opening its scope to a new range of applications. Ultra-low latency is one of the requirements of such applications in the next generation of mobile networks (5G), where deep learning is expected to play a big role. Hence, to enable the usage of deep learning solutions on the edge cloud, ultra-low latency inference must be investigated.

The study presented here relies on the usage of an in-house framework (CRUN) that enables the distribution of acceleration on data center environment. The objective of this thesis is to leverage the best solution for the inference of a machine learning algorithm for an anomaly detection application using neural networks in the edge cloud context. To evaluate the obtained results with CRUN a comparison work is also carried out. Five inference solutions were compared using CPU, GPU and FPGA.

The results show a superior performance in terms of latency for all CRUN experiments, that basically comprehends three cases. The first one utilizing the RTL anomaly detection neural network as a baseline solution, the second using the same baseline code but unrolling the biggest layer for obtaining reduced latency and the third by distributing the neural network in two FPGAs. The requirements for this solution were to obtain latency between 20μs to 40μs for inference time and at least 20 000 inferences per second. These goals were categorically fulfilled for all CRUN experiments, providing 30 μs latency in average, while the second best solution provided 272 μs.

PREFACE

First of all, I thank God for my life and all the blessings that I have received.

This thesis work is part of a bigger project at Nokia. I would like to express my gratitude to the company for the opportunity of developing and writing my thesis, even during working hours. This support was immeasurable.

A big thank you goes to my supervisor Prof. Timo Hämäläinen for the guidance during the entire thesis writing.

I would like also to thank my colleagues in Nokia for the support and advices provided for the writing of this thesis. In special, my sincere acknowledgment goes to Petri Kärppä for valuable technical and academic guidance during this work. Also, my profound gratitude to Jouni Siirtola for the technical background and restless help with the work carried out here. I am grateful to Jouni Markunmäki for guidance and Juho Tieaho for the cooperation during this thesis work. Additionally big thanks go to Pekka Jokela, Hannu Tulla, Anssi Örn and Kalle Holma.

I deeply appreciate the support of my family, in special, my mother Nilceia de Fátima Tobias Carneiro, whose love and care has been the foundation of my life and my late father Jorge Tobias Carneiro who never, even in his wildest dreams would have thought that his own daughter would graduate in Finland.

Finally, I wish to give my deepest thank you to my partner in all aspects of my life Daniel Koslopp, whose love and support keep me alive.

Tampere, 18.11.2018

Talita Tobias Carneiro

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

C-RAN	Cloud-based Radio Access Network
ACAP	Adaptive Compute Acceleration Platform
AHU	Accelerator Hardware Unit
API	Application Programming Interface
AR	Augmented Reality
ASIC	Application-specific Integrated Circuit
AWS	Amazon Web Services
BBU	Baseband Unit
BLAS	Basic Linear Algebra Routines
BNN	Binarized Neural Networks
CNN	Convolutional Neural Networks
COTS	Commercial Off-The-Shelf
CPRI	Common Public Radio Interface
CPU	Central Processing Unit
CPU-1	CPU batch-1
CPU-16	CPU batch-16
CRUN-B	CRUN Baseline
CRUN-D	CRUN Distributed
CRUN-U	CRUN Unrolled
CU	Central Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DNN	Deep Neural Networks
DPDK	Data Plane Development Kit
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
DU	Distributed Unit
FaaS	FPGA-as-a-service
FFT	Fast Fourier Transform
FIFO	First In First Out
FLOP	Floating-point Operations
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GEMM	General Matrix Multiplication
GEMX-32	GEMX batch-32
GPU	Graphics Processing Unit

GPU-1	GPU batch-1
GPU-16	GPU batch-16
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLS	High-Level Synthesis
ILA	Integrated Logic Analyzer
IoT	Internet of Things
IP	Internet Protocol
ISA	Instruction Set Architecture
LSVRC	Large Scale Visual Recognition Challenge
MAC	Multiply-Accumulate
MEC	Multi-Access Edge Computing
ML	Machine Learning
MLP	Multi-Layer Perceptron
NFV	Network Function Virtualization
NN	Neural Networks
NPU	Neural Processing Unit
NRT	Non-Real Time
NVDLA	NVIDIA Deep Learning Accelerator
PCIe	Peripheral Component Interconnect Express
PE	Processing Element
QNN	Quantized Neural Networks
QoE	Quality of Experience
RDMA	Remote Direct Memory Access
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Networks
ROM	Read-Only Memory
RRH	Remote Radio Head
RRU	Remote Radio Unit
RTL	Register Transfer Level
SDAccel-1	SDAccel batch-1
SDAccel-16	SDAccel batch-16
SDN	Software-Defined Networking
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SoC	System on a Chip
TNN	Ternary Neural Networks
TPU	Tensor Processing Unit
TTI	Transmission Time Interval

VM	Virtual Machine
vRAN	virtualized RAN

a	neural network activations
$\mathbf{A}$	matrix A
α	scalar for matrix multiplication
b	biases
$\mathbf{B}$	matrix B
β	scalar for matrix multiplication
$\mathbf{C}$	matrix C
$f(x)$	inference computation
f	prediction function
fg	function to be approximated
$h(.)$	differentiable nonlinear activation function
op(.)	original or transposed matrix
σ	sigmoidal output unit activation function
w	neural network weights
x	neural network inputs
y	neural network outputs
z	second layer of the neural network

1. INTRODUCTION

It is well known that the next generation of mobile networks must support an ever increasing number of mobile data traffic. It is only true that mobile communications are the world's largest technology platform [9].

The great capacity demands can only be answered with a massive evolution in mobile networks architecture. The latest trend in this context is the centralization of baseband functions that were once performed in a distributed fashion, usually very close to the antenna site [40]. This effort is done in order to provide flexibility and dynamic scalability for future applications. It also has an interest in running baseband functions not only centrally but in a virtualized environment, so commodity server hardware can be used. This architecture is commonly referred as Cloud-based Radio Access Network (or C-RAN) or as industry seems to prefer virtualized RAN (vRAN).

In order to use commodity server hardware in this context, the principles of Network Function Virtualization (NFV) must be used. However, as more and more demanding functions are virtualized, it gets more and more difficult to attend their requirements, in terms of latency and throughput for example, with Commercial Off-The-Shelf (COTS) hardware and hardware acceleration must be employed.

A promising approach is to utilize neural networks in C-RAN. Deep learning has a tremendous power when it comes to its applications. The appeal of deep neural network solutions is undeniable. Since the dramatical reduction of the error rate in the Large Scale Visual Recognition Challenge (LSVRC) [43], the interest in this field has been renewed. Machine learning has proved to be an excellent tool, part of this success is boosted exactly by Deep Neural Networks (DNNs). The major breakthroughs experienced during the last decade, especially in computer vision and natural language processing are a result of this field of research.

The DNNs ability of automatic feature extraction differs from the hand-made features or rules devised by experts and is the reason why these solutions achieve such superior performance when compared to other techniques. The algorithm is able to

learn statistically from a large dataset the representation of the input space. This learning phase is referred as training, in which a set of examples must be used to adjust the weights that forms the model.

Once a DNN is trained, it is ready for use, so inference phase can start. These two distinct phases have different computational demands. Training requires throughput while inference is concerned with latency. In this sense, the first presents higher computational workload and is a fit for GPUs (Graphics Processing Unit), but the second although being carried out in CPUs (Central Processing Unit) and GPUs alike has gaining a rising interest for a specialized solution. The number of application specific processors for ML (Machine Learning) inference offered by industry increases all the time. These solutions come from tech giants and start-ups, in which some examples are Tensor Processing Unit (TPU) from Google, Myriad from Intel, Huawei, Cerebras, Groq and others.

In this context, industry's interest is towards faster inference rather than training. But why? It is possible to deduce that inference is the production step with DNNs. Indeed, it is only after training that the model is deployed to deliver the required predictions.

Important to highlight that, since training takes a huge amount of time and inference is where the model is put to a use, that the interest on industry is to accelerate and make better and faster predictions. Either to support the development of the Internet of Things (IoT) on the device or to boost the extensive set of applications on cloud data centers.

With this in mind, it is natural that DNN solutions can be effectively applied across the entire mobile networks architecture. However, an interesting point in Cloud RAN is in the edge cloud concept, in which the idea is to distribute cloud capabilities across the network placing computing resources at the edge of the network. In this scenario, a myriad of applications can be accelerated, ranging from baseband functions, management scope and analytics applications with the Multi-Access Edge Computing (MEC). For example, auto-encoders can be used for anomaly detection problems, a common application in mobile networks.

In order to enable the usage of deep learning inference applications in the edge cloud, one needs to investigate how to minimize the latency of such algorithms in a system level and on the applications level. Latency is important for deep learning inference, it is even more important and indisputable in the edge cloud context.

Within this scenario lies the exact goals of this thesis, the investigation of possibilities

vs. requirements. In special, the study of the usage of an in-house framework (CRUN) that enables the distribution of acceleration on data center environment. The objective of this thesis work is to leverage the best solution for the inference of a machine learning algorithm for an anomaly detection application. The requirements for this solution are ultra-low latency between 20μs to 40μs for inference time and at least 20 000 inferences per second. Thus, the fifth generation of cellular networks latency demands can be fulfilled.

A comparison work is the final contribution of this exploration between five implementations, from GPPs (General Purpose Processors) architecture to a hand-optimized RTL (Register Transfer Level) neural network implementation allied with CRUN framework.

This thesis work is structured as follows. Chapter 2 discusses cloud computing in mobile networks scope, its requirements and the role of deep learning in this context, placing this thesis on this domain. In Chapter 3 the basics of deep neural networks is reviewed, presenting important concepts and the considerations when choosing a hardware platform for implementation. Following this first brief introduction to neural networks, the examination of the possible optimizations is done from a software perspective. An overview of hardware implementations of inference accelerators and the closest related works is done in Chapter 4, in which the review is divided into optimizations, tools and architectures propositions and the use of such designs in cloud computing. The methodology of this thesis is presented in Chapter 5. Subsequently, Chapter 6 summarizes the in-house infrastructure used to accelerate the example NN application over the Ethernet, the optimizing techniques utilized and the software application used for running the system. Finally, the results are presented in Chapter 7 and the comparison between different platforms and network size is discussed, as well as the considerations of the distribution of the application. Conclusions and next steps are outlined in Chapter 8.

2. MOBILE NETWORKS AND CLOUD COMPUTING

The domain of this thesis work is mobile networks and its evolution towards centralization and cloudification. In special, the study of how deep learning applications can be accelerated on the edge of the network.

The needs of neural networks are diverse. In general, one of the main reasons for scaling up machine learning solutions is the inference timing constraints, which requires predictions to be made in real-time [5]. From the computational load perspective, only the inference is the object of study in this work. In this sense, its computation can be either done on the cloud or on the device [70].

The application requirements and its scope will dictate where the inference of the neural network will be processed. From the cloud viewpoint, the importance of the latency requirement is becoming crucial as applications with live streams get more and more popular for cloud service providers [17].

However, in order to fully understand the requirements which cloud computing imposes, one must first comprehend its scope. In this respect, the National Institute of Standards and Technology (NIST) definition for cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [51, p. 2].

In Cloud computing-based architecture for Radio Access Networks (C-RAN) there are the same interests to accelerate neural network inference workload as in traditional cloud. The tight requirements on latency appeal to a similar approach. This chapter introduces C-RAN concepts and main requirements.

2.1 C-RAN

C-RAN is one of the answers for the continuous growth experienced in mobile data traffic [13]. The surge observed in mobile data transmission can be explained by the ever increasing number of smartphones and applications [83]. To put it in numbers, according to [18] the 49 exabytes (10^{12} MB) mark will be reached monthly by 2021.

The C-RAN concept addresses the challenges of adapting to a non-uniform traffic and the efficient resource utilization. In this sense, it comprises a novel mobile network architecture [13], that has evolved from the traditional distributed approach in which a base station was responsible for the baseband and the radio processing [40]. The nomenclature used can vary, but it is common to refer to the radio processing portion as Remote Radio Head (RRH) and the Baseband Unit as BBU [13].

The evolution towards C-RAN starts with the location in which RRH and BBU are placed. In a traditional distributed architecture, they can either be located at the base of the cell tower with a coaxial cable connection to the antennas, or in a split manner in which RRH is at the top of the cell tower with the antennas and the BBU is in a nearby cabinet with a fiber connection between the two [66]. From Figure 2.1 (a) the traditional approach is showed; the figure depicts the split architecture only.

The keynote in C-RAN is the capacity of centralizing the baseband processing and share its resources in a virtualized BBU pool [13], which can also be referred as vRAN. This ability employs a sophisticated communication and cooperation mechanisms between base stations [40].

In this architecture, the baseband processing units (BBUs) consist of the central pool of resources, the communication with different base stations must be made with low latencies and high throughput. The BBU pool enables the dynamic allocation of baseband processing resources to different cell sites and radio technologies [66]. The radio signals are collected from distributed antennas into remote radio heads in which they are transmitted through optical transmission network to the cloud platform [40].

Figure 2.1 shows the differences between the traditional and the C-RAN architecture. Note the centralized BBU as the resource pool. Additionally, observe the three main parts of this architecture, the RRH, BBU and the Fronthaul connections, in which the last one connects the other two components [40]. On another perspective, the Backhaul connects the BBU to the mobile core network [13]. Refer to Figure

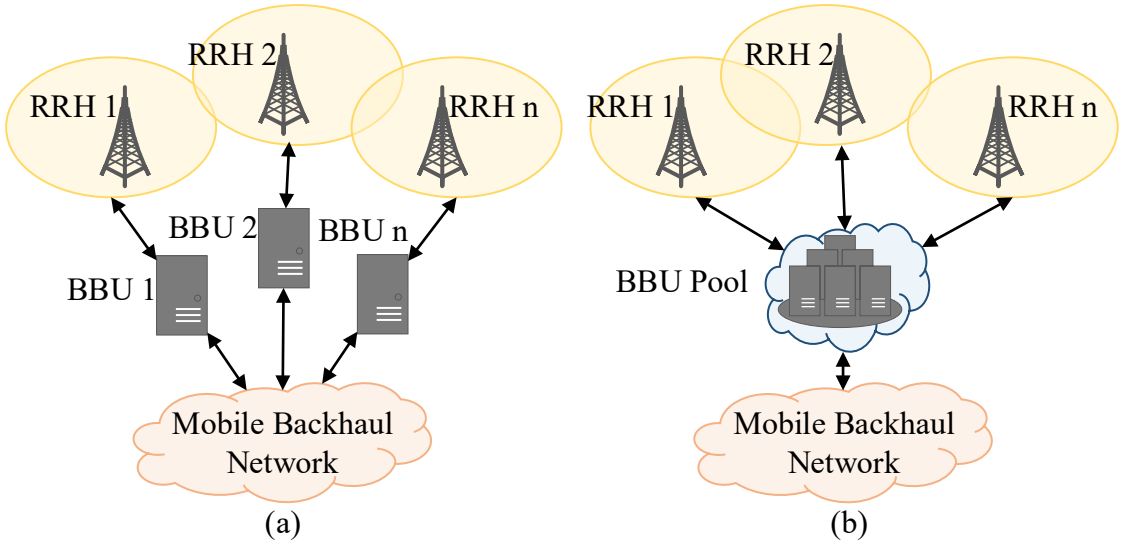

Figure 2.1 *Distributed BBU and C-RAN architecture. Adapted from [13].*

2.2 for Fronthaul and Backhaul connections.

The full potential of a centralized architecture can be achieved with the virtualization of BBUs. The vRAN architecture utilizes vBBUs (virtualized BBUs) deployed in centralized data centers and the RRHs still remain at the cell sites on the edge [66]. The concept is still fuzzy regarding its name, this virtualized approach could still be referred to as a C-RAN implementation.

In the C-RAN context, the baseband unit is deployed centrally at a network-edge data center [9]. In Figure 2.2 these components correspond to the BBU pool location. These facilities, when designed using cloud principles, provides the opportunity of running also multi-access edge computing (MEC) services [9]. Moreover, the RAN edge offers an ultra-low latency and high-bandwidth with real-time radio network information environment that can be used by applications and services [62].

From a cost perspective the deployment of MEC and C-RAN should be done as one. As such, since the BBU pool is already planned in C-RAN, the cost of providing additional processing (MEC) in the same facilities is lowered [65].

Considering this, C-RAN model offers an integration possibility between radio access with the rest of the telco cloud-enabled network, in which the same edge data center hosts the application logic or content on cloud infrastructure and the centralized control functions [9]. As of the writing of this work, it can be formalized, that in 5G there is a Central Unit (CU), a Distributed Unit (DU) and the Remote Radio

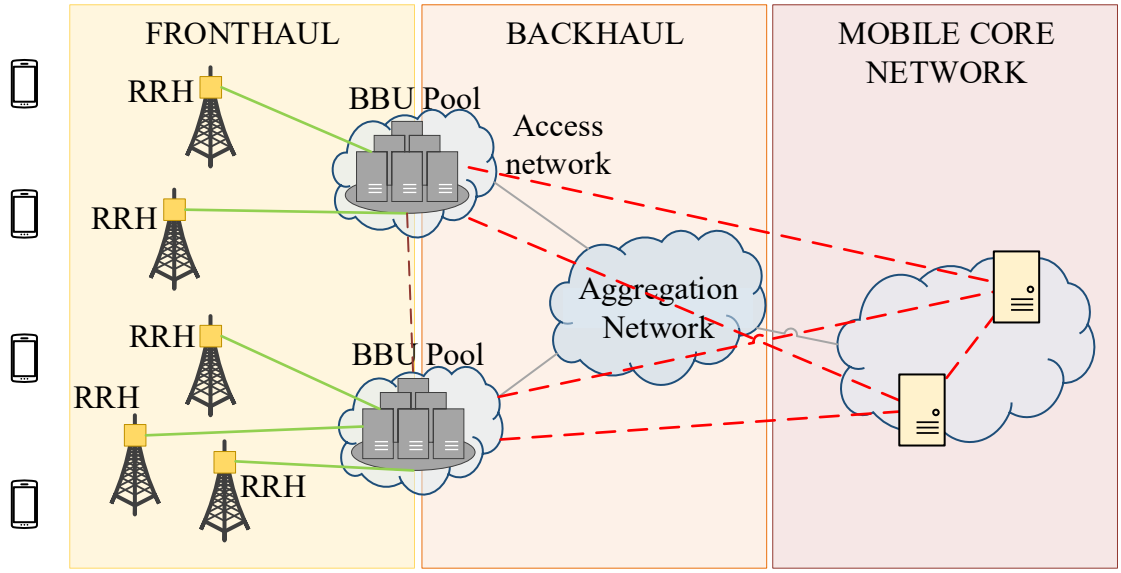

Figure 2.2 *C-RAN mobile network. Adapted from [13].*

Unit (RRU), which in 4G/LTE corresponded to the original BBU function. From the deployment point of view, this leads to several options, in which each scenario depends where each unit is located, for example the evolution from single-node in 4G to split function architecture of 5G [35].

From the function perspective, an important question emerges, what is the best functional split between these units. Academia and industry alike are concerned with the best trade-off and several propositions are made, including a flexible approach, as described by [32, 12, 50, 11].

The interest in this functional split is manifold. The reason is that as the data rates increase the conventional fronthaul implementation (using Common Public Radio Interface - CPRI) is impractical [35]. Figure 2.3 shows the optional split points.

In order to best choose the optimal split point, the trade-offs between throughput, latency and functional centralization must be taken into account. Observing Figure 2.3 it is possible to infer that moving towards a higher layer split, left-side of the picture, means fewer processing functions to be centralized but relaxed requirements when considering throughput and latency [35].

Another view to the same problem is whether to split Real-Time (RT) functions and Non-Real-Time (NRT) ones. In this sense, the former would be deployed at the antenna site for air interface resources management and the latter control functions

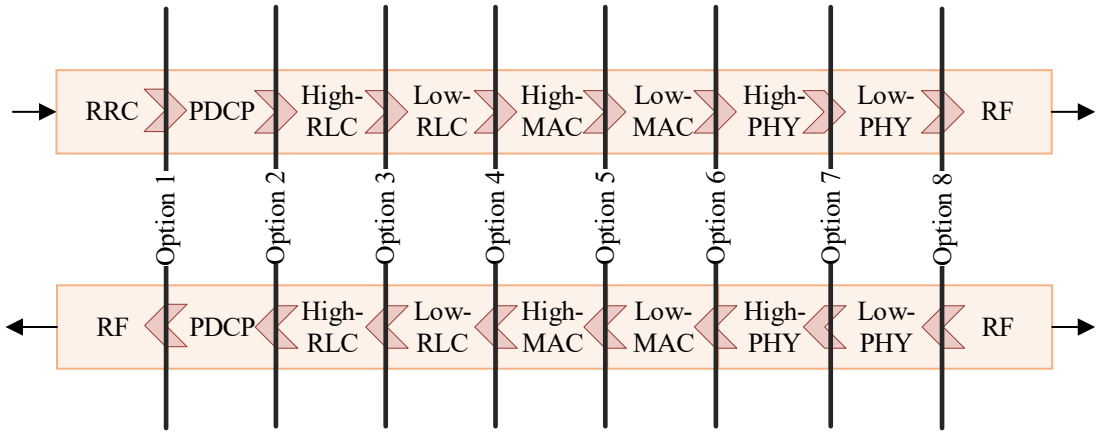

Figure 2.3 *Optional split points. Adapted from [35].*

hosted centrally [9]. In essence, the best performance gains would be observed if the entire protocol is centrally controlled, in Figure 2.3 this means Option 8. Consequently, the requirements between CU and DU would be ultra-low latency and high-bandwidth.

In order to emphasize the requirements and the challenges that they impose, table 2.1 shows the contrast between cloud computing requirements and C-RAN. It is important to highlight that this work is mainly concerned with latency, data profile and data rate, in this order of importance.

Table 2.1 *Comparison between Cloud computing and C-RAN requirements [13].*

	Cloud Computing	**C-RAN**
Data rate	Mbps range	Gbps range
Data profile	Bursts and low activity	Constant stream
Latency	Tens of ms	Hundreds of μs
Jitter	Tens of ms	ns range
Information Life time	Long (content data)	Extremely short
Recovery time	s range	ms range
Number of clients	Thousands to millions	Tens to hundreds

2.2 NFV and SDN

In a broader scope, C-RAN is a use case of Network Function Virtualization [33]. NFV describes a technique in which the network functions, traditionally computed in specific network hardware (i.e. bare metal), are run as application software in a general infrastructure hardware. In order to achieve this end result, a virtualization layer (hypervisor) is used for virtualizing the physical hardware resources as computing, storage and network [40].

Indeed, note how NFV is one key enabler for C-RAN architecture in this sense. Also, how the deployment of MEC and C-RAN can be carried out as one in this context. As such, in the vRAN model, the deployment of vBBUs is done on multiple NFV platforms utilizing standard x86 hardware and consolidated in central data centers [66]. A second essential concept in this scope is Software Defined Networks (SDN), which is intrinsically related to NFV. According to [53], a networking solution that combines NFV and SDN leads to a greater value resource.

SDN is a networking paradigm that provides centralized control of the network. It eases the separation of the control and data plane [21]. As a result, networks are programmable, adaptable and cost effective [40].

The decoupling of control plane and data plane is an important concept, since traditionally they were packaged into proprietary, integrated code from proprietary vendors [40]. This shift in abstraction shapes the functionality of network switches, in which they become dummy packet forwarding devices that are controlled logically by a centralized entity [13].

So far, the concepts behind cloud computing, NFV, SDN and C-RAN were introduced. Although each one of these fields seem to relate, no clear relationship was established. For that purpose, in this work, the relationship proposed by [53] is adopted. In this sense, NFV, SDN and cloud computing are abstraction of different resources, where compute is for cloud computing, network for SDN, and function for NFV [53]. As for C-RAN, it can be understood as an example of this resource abstraction endeavor.

2.3 Deep Learning in Mobile Networks

The fields of deep learning and mobile networks have mainly been researched separately [83]. However, recently the emergence of a combination of these two research disciplines can be observed. For a comprehensive survey on this topic, the reader should refer to [83].

In the evolution of mobile networks, the road leads for the 5^{th} generation (5G) of mobile systems. As such, the fifth-generation technologies, namely full-duplex, ultra-dense networks and large scale antenna systems can be facilitated in full scale by the flexibility and scalability that only a cloud-based approach as C-RAN naturally imposes [40].

Deep learning has a wide range of applications. The same assertion is true in the

scope of mobile networks [83]. To the extent of this work, the deep learning applications considered here are the ones pertinent to the edge cloud concept, introduced in sub Section 2.1.

In this context, the edge of a mobile network is not only intended for specialized processing, as it was in the past. It now offers the possibility to integrate applications with radio equipment enabling a new set of high value services [62]. Consequently, in this scenario a broad range of applications can benefit from deep learning solutions.

In the management level utilizing network-level data, as an example, the work carried-out by [63] demonstrates the use of MLPs (Multi-Layer Perceptron) for user's QoE (Quality of Experience) prediction by using average user throughput, number of active users in a cell and channel quality indicators.

Some use cases are depicted as examples in the edge cloud scenario in [62], at least two are a perfect fit for deep learning, especially CNNs (Convolutional Neural Networks), cited here:

- Augmented reality content delivery: the edge data center can provide applications performing local object tracking and local AR (Augmented Reality) content caching,
- Video analytics: by processing the video stored by the video management application to detect and notify specific configurable events.

The closer to the edge the tighter latency requirements. The mapping between use case latency and the different levels of distributed data center as possible location can be referred from [65] and [9]. Important to highlight that because the 5G central unit is also deployed in the edge data center, it is suitable for very low-latency services, as it is the case of assisted driving, which again comprehends an important deep learning application.

There is still a crucial characteristic when it comes to edge cloud responsibilities. Depending on the functional split between CU and DUs, there is more room for lower layer processing into edge data centers. Again, refer to Figure 2.3, according to [35], the choice of the optimal split point depends on the specific deployment scenario.

Figure 2.4 shows the base station functionalities separated into BBU and RRH, although no separation for CU and DU is done. Note the wide scope for opportunities of using deep learning solutions for L1, L2 and L3 processing.

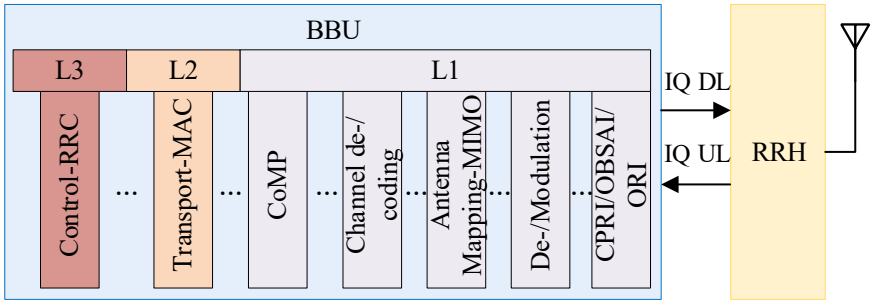

Figure 2.4 *Base Station functionalities. Adapted from [13].*

As an example, the investigation proposed in [82] shows the usage of deep learning for channel estimation and symbol detection since DNNs present the ability for learning and analyzing characteristics of wireless channels suffering from nonlinear distortion, interference and frequency selectivity. Similar objectives were investigated in the work proposed by [54] using different machine learning approaches. This application requires very short latency and referring to Figure 2.4 would be mapped to L1 processing.

The proposition done by [60] is to interpret a communications system as an autoencoder, opening the view for the use of deep learning with the physical layer, L1 on Figure 2.4.

Finally, the efforts in using machine learning, especially deep learning, in 5G networks is also driven by industry. In [15], three examples are mentioned: beamforming scheduling; indoor positioning and downlink/uplink channel configuration.

The beamforming technology makes it possible to transmit beams of data for targeted users, which minimizes interference and efficiently uses the radiofrequency spectrum. One potential issue in using beamforming is the scheduling of such beams, a combination problem of four out of 32 beams gives 30 000 options. The usage of deep neural networks for implementing this scheduler was already claimed by industry [15]. This application corresponds to L2 in Figure 2.4. With this in mind, it is possible to infer that the latency requirements for such solution is indeed ultra-low, in the extreme case every sub air interface Transmission Time Interval (TTI) corresponding to some tens of microseconds for the ML inference response.

In addition, the interested reader could refer to [36] for a bigger overview of the usage of machine learning techniques as a whole in wireless communications, not only focusing on deep learning.

In this thesis work, a real-time anomaly detection neural network for mobile network

traffic is considered. The application corresponds to the management level of mobile networks and can be mapped to the edge cloud context that in C-RAN could be deployed in the BBU Pool.

3. NEURAL NETWORKS

Machine learning aims to create algorithms for making predictions based on data. In this sense, the mapping between input to output is the main task of such algorithm which is a predictive function [5].

The main division for machine learning algorithms is the nature of the training phase. There are two basic approaches, supervised learning and unsupervised learning. Supervised learning must utilize a set of training data for constructing the prediction function and apply it to the test data. The typical format for the training data is labeled examples, which comprises the data instance and the ground truth [5].

In contrast, unsupervised learning operates in a set of inputs without any labeling corresponding to it. In this case, the goal differs from supervised learning since some sense must be made from the unlabeled data [6]

Artificial neural networks (or as they commonly are referred nowadays, only neural networks) are inspired in biological structures. They are basically an attempt to model the biological information processing of the nervous system [67]. Modern NNs (Neural Networks), however, should not be understood as an accurate model for the brain, but instead as function approximation engines which the basic underlying ideas are borrowed from neuroscience [27].

In this sense, the hierarchical multi layered structure sets the pace for the transmission of information for neighbor's units and more distant ones. It is important to highlight that the parallel computation is a nature aspect in neural networks. [67].

Feedforward neural networks, also called Multi-Layer Perceptrons (MLP) are the foundation of deep learning. For a classification problem, an MLP establishes the mapping between the input and the class which the input belongs. Note from the name, feedforward, that the output of the model is not fed back into it. But this does not mean that it is not possible. In this case, the neural network is called a Recurrent Neural Network (RNN). RNNs present state of the art predictions for speech recognition tasks, for example [27]. A third common type of NNs are the Convolutional Neural Networks (CNN) that are specific MLP types with a convolutional

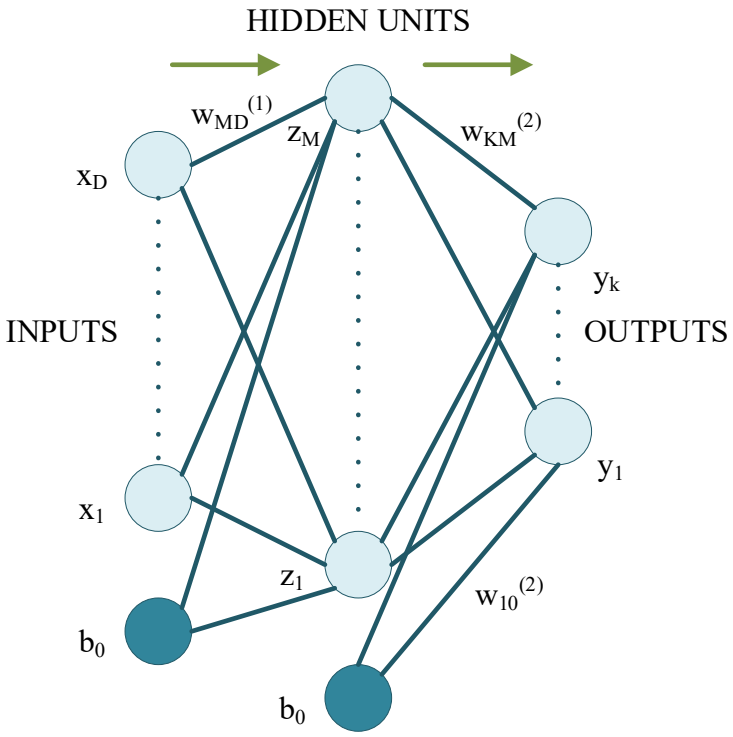

Figure 3.1 *Basic two-layer neural network diagram. Adapted from [6].*

layer intended for feature extraction. This set of neural networks are important for vision tasks and object recognition.

3.1 Mathematical Definition

Figure 3.1 depicts the basic two-layer neural network diagram and shows the basic building blocks of a neural network.

Jumping to mathematics, one can define feedforward networks from its fellow linear models for classification and regression. For a walk-through of this process, refer to [6].

The starting point to devise the basic neural network model is given in equation 3.1, where w corresponds to the weights, b to the biases. Note the superscript (1) which corresponds to the layer of the network (in this case, the first layer). The activations correlate with the quantity a [6]. Again, refer to Figure 3.1 for reference.

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)} \tag{3.1}$$

Each of the activations expressed in 3.1 are transformed by a differentiable nonlinear activation function $h(.)$, here given by equation 3.2.

$$z_j = h(a_j) \tag{3.2}$$

From Figure 3.1 observe the correlation with equation 3.1 and 3.2. The final network function is provided in 3.4.

The process is repeated again by linearly combining the results in z, which corresponds to the second layer of the network, this can be seen from Figure 3.1. From equation 3.3 below, b corresponds to the bias.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + b_k^{(2)} \tag{3.3}$$

A feedforward neural network is, in this sense, a series of functional transformations [6]. The expression in 3.4 shows the final form of a two-layer neural network model, where y_k give the set of network outputs and σ represents the sigmoidal output unit activation function, which can be used for binary classification problems. Note the matrix-matrix multiplications on equation 3.4.

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma(\sum_{j=1}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)}) + b_k^{(2)}) \tag{3.4}$$

The process to choose the activation function to be used is determined by the nature of the data and follows a specific set of rules [6]. The most used activation function nowadays is the Rectified Linear Unit (ReLU), depicted in Figure 3.2. The main advice is to use ReLU as the activation function for modern neural networks [27].

This recommendation comes from the fact that ReLU is a piecewise linear function composed of two linear pieces, which makes it almost linear and as such, the optimization with gradient descent methods is straightforward [27].

In essence, the nonlinearity inserted with non-linear activation functions between fully connected layers is necessary, otherwise a multi-layer network could be arithmetically minimized to a one-layer deep neural network.

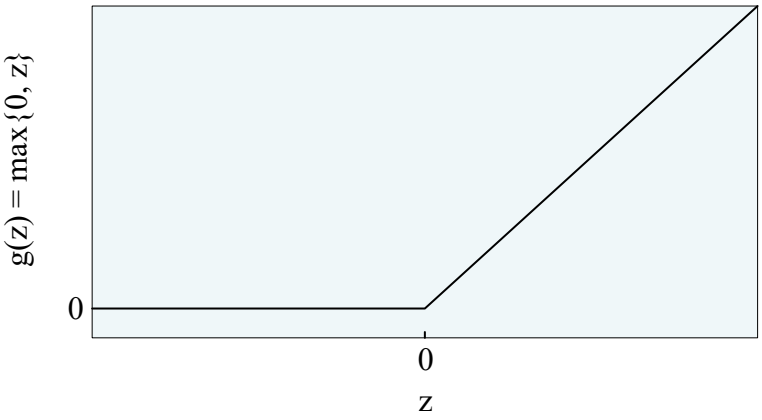

***Figure** 3.2 Rectified linear (ReLU) activation function. Adapted from [27].*

3.2 Concept Definitions

In a neural network, the first layer is commonly called the input layer, similarly the last layer is the output layer. The layers in the middle of the neural network are called the hidden layers and their relationship to the network is tightly related to training. [27]. Once again, acknowledge Figure 3.1 for reference.

Since the goal of a feedforward network is to approximate a given function fg, during the training phase the objective does not change. The training data does not specify the behavior of the hidden layers, instead the important concept here is that the hidden layers must be used at its best to approximate the function fg. Thus, simply put, the hidden layers are called as such because the output provided by them is not yet the desired approximation. [27]

The depth of the model is given by how many layers the neural network has, which directly correlates to how many processing stages it contains. From these two affirmatives, it is possible to explain two important terms. The term deep for deep learning comes from the model depth. On the other hand, multi-layer perceptron, comes from the fact that each layer resembles the perceptron model [27]. For more information on this model refer to [6].

It is important to highlight that a perceptron is only one of the many artificial neuron models proposed in 1950s and 1960s [55].

3.3 Training and Inference

Deep neural networks are deployed in two phases: training and inference. Simply put, training regards to identifying the prediction function f, while inference com-

putes $f(x)$ on a data instance x. In order to understand the computational demands of these two tasks, one must get a glimpse of the underlying concepts regarding how neural networks are trained.

Without considering the specifics of training, one can assume that it basically means an iterative procedure in which the objective is to minimize an error function by adjusting the weights in a sequence of steps. At first, the evaluation of the error function derivative with respect to the weights is carried out. In the subsequent step the weights are to be adjusted accordingly with the derivatives evaluated at the previous step. There is a distinction in these two steps, and as such different techniques are considered for each. In this regard, back-propagation can be observed for the former whilst gradient descent for the latter. Notice that many more powerful optimization techniques can be used instead of gradient descent, and its mention is just an example due to its simplest form [6]. Another technique may be stochastic gradient descent.

In a feedforward neural network, forward propagation consists of the input x being propagated through the hidden layers until it reaches the output layer producing y. In contrast, during the training phase, the back-propagation algorithm is responsible for the flowing of information backwards in the network from the cost for gradient computation in which the cost is a result of the forward propagation during training. [27]

The process of training a neural network is highly computational intensive, the task of iteratively calculating gradients and adjusting weights until the labeled data is correctly predicted is indeed exhaustive. On the other hand, when compared to the inference stage, one can only argue that inference offers a much easier task, since only the forward pass takes place.

The previous remarks are the only ones referring to training and learning step on NNs. For the remaining of the present work, all the investigation regarding neural networks will be only focused on the inference phase of deployment.

3.4 Inference's Computational Load

The main achievements observed today, when referring to neural networks astonishing performance in certain applications, are basically due to two main reasons: the increasing computing power and the abundance of data. Analyzing over the last twenty years, the growth in network sizes is exponential [23]. Early networks would follow this premise, VGG would have around 2x the size of AlexNet that already

had 60M parameters [57].

It is true though that when analyzing a shorter period of time, the challenge of maintaining bearable computational workload while increasing the accuracy has been tackled by recent model approaches [2]. Consequently, these more recent DNNs are especially designed to be more efficient since there is a trend that the deeper a neural network is, the more accuracy it will provide [57].

When referring to computational workload, CNNs are the most studied subject for inference accelerators, as can be seen in [52, 48, 64, 85, 61, 22].

A CNN model comprises of convolutional layers and fully connected layers. The complexity and computational requirements of these two layers types are different. A convolutional layer is computation-centric while a fully connected layer is memory-centric. Furthermore, the former uses few parameters but needs heavy number of operations while the latter utilizes hundreds of millions of weights that are used for one time only [64].

An efficient CNN inference accelerator would take this unbalance in computation to memory-ratio and apply different techniques for each portion of the neural network. Although this work focuses on a fully connected layers network, the revision of strategies will be made in a general format along with the trends in efficient design.

There are many techniques for accelerating inference for CNNs. From the software standpoint, the goal is to compress the model, reducing the memory footprint, the number of operations while trying to maintain accuracy. On the other hand, from the hardware perspective, the objective is to design the architecture to reuse data as much as possible, increase its locality and accelerate the convolution operations. Additionally, reducing the precision is also a target for efficiently deploying these models [64].

3.5 Execution Platforms

Before going deeper in the specifics of these strategies, the next paragraphs state the most important points considering GPPs (General Purpose Processors), FPGAs (Field-Programmable Gate Arrays) and ASICs (Application-Specific Integrated Circuit). For a review on related work in FPGA-based inference accelerators, please refer to the Chapter 4.

Traditional general-purpose architectures are usually the choice platform for training and predicting neural networks. In order to meet performance requirements, when

talking about CPUs, they are more likely to be used in large clusters [46]. GPUs on the other hand, are well known for performing data parallel computation with high throughput for floating point in regular parallelism. However, even when increasing the number of Floating Point Operations Per Second (FLOPS/s), GPUs support only a set of native data types, which essentially means that for custom data types it may perform poorly [57]. When comparing CPU clusters with GPU, the former waste a big portion of its resourcing with synchronization between cores. Furthermore, in applications which the memory transactions are small when compared to arithmetic operations, GPUs are a better choice [46].

From a memory point-of-view, general purpose processors rely on traditional Von Neumann architecture. This means that instructions and data are stored in external memory and are fetched when needed by the software execution. The motivation for memory hierarchy lies exactly in this fact, for reducing the costly external memory operations. In this regard, however big performance a GPP may offer, the memory-processor communication is the bottleneck in such architectures. This together with the costly memory-bound deep learning operations makes the GPP performance to suffer irrecoverably [44].

When it comes to ASICs, their specific purpose nature is converted into limited programmability [56]. Although they typically provide the highest performance and energy efficiency with the smallest chip size, their design takes a substantial amount of time. In addition, after the tape out of the chip, inserting new features or finding design errors translates into a new set of masks and considerably more time for a new process. In this sense, they are only used with applications that requires a high volume of these chips, so the effect of the cost can be diminished [86].

Two important factors contribute to the advantages of FPGAs as inference's execution platform. Firstly, an FPGA device, with its reconfigurable logic offers the possibility of using different and custom data types. Secondly, they can utilize the distributed on-chip memory and pipelining, which means a great deal in feed-forward systems. Also, the possibility of partial dynamic reconfiguration plays a central role in architecture planning. But the irrefutable truth is the level of solutions tailoring, with extreme freedom for exploring optimizations [44].

3.6 Network Model Optimizations

In general, the usage of floating points, although well supported by GPPs, is not an efficient implementation in ASICs and FPGAs, which are much more efficient when using fixed-point arithmetic [64]. Avoiding floating point operations is a reasonable

approach in the DNN context.

Data quantization is one of the most common methods for reducing the precision of activation values and weights without having a heavy impact in prediction accuracy [2].

The benefit of using quantization is twofold. Firstly, the use of less bits will reduce the memory footprint, its bandwidth and storage requirements. Secondly, the adoption of simpler representation will reduce the hardware cost in the operations standpoint [30].

At least two different approaches for quantization can be identified from literature, static fixed point and dynamic fixed point. In the first, the bit-width is set according to the numerical range and the precision required, thus every operand share the same scaling factor. Each number is then quantized to the nearest fixed-point representation. One identifiable problem with this approach is that the dynamic range of floating point representation is much bigger than the fixed point data, which yields in either overflow or underflow [30]. To address the problem of the first, in the second approach type, the scaling factor can differ according to the parts of the network. This is due to the fact that separate portions of a network can have different numerical range of data [2].

In addition, quantization is a method that can offer various flavors combinations. Indeed, when referring to quantized inference, the phase in which quantization is applied is also an important factor. If the objective is to reduce model size without the need of retraining the model, then post training quantization is an option, which is a simpler method yielding good results. However, when aiming at higher accuracies quantization aware training should be considered. For more information about quantization for efficient inference refer to [42].

It is indeed a trend in deep neural networks to improve efficiency by taking into use compact data types, even with floating point representation. According to [57] the usage of below 32-bit single precision floating point is the new norm.

Recently, research efforts have been directed to study the usage of extremely compact data type representation, a big portion of these works refer to Binarized Neural Networks (BNNs). These networks are proposed on the basis of using 1-bit representation for neurons and weights, in which values are constrained to +1 and -1 [57]. The impact of using this representation on FPGAs is huge. It essentially means that the multiply-accumulate (MAC) operations can be mapped to XNOR gates followed by a bit counting operation. It is irrefutable though that the performance gained

with this method is heavily translated in accuracy degradation [2].

The work proposed by [72] targets binarization for all input activations, weights and output activations. A second generation of the same proposition is done in [7], in which the support for mixed and variable precision is added, as such, it targets a bigger scope not only BNNs but also QNNs (Quantized Neural Networks).

Another effort targeting BNNs is XNOR Neural Engine [20], which is a hardware accelerator IP integrated within a microcontroller unit for low-power solution on the device.

Along the same path as BNNs, one can also find the ternary neural networks (TNNs), in these type of networks, the weights are represented by 2-bit values and are constrained to 0, +1 or -1. In cases in which there is negligible accuracy loss the neurons are not quantized.

If on one hand data quantization can be effectively used for optimization of neural network models, on the other hand the number of neurons and weights can also be optimized for efficiency purposes.

In this context, pruning is a method that relies on exploiting sparsity (i.e. the near zero values) in neurons and weights [57]. In fact, DNNs are often over-parametrized in the sense that a big portion of its parameters can be pruned because they are redundant [81]. The importance and applicability of this optimization has grown in the recent years, this is due to the broad usage of ReLU as activation function, which zeros out negative values. Consequently, the sparser a matrix the fewer operations needed for its computation [57]. The pruning for weights is also very relevant [2]. The values that are zero out are interpreted as not important and this approach can maintain the original accuracy [57].

One of the drawbacks of pruning is the irregular resultant network structure. Targeting only CNNs, CirCNN [22] presents the usage of block-circulant matrices for representing weights which reduces the storage and computational complexity without pruning.

It must be kept in mind though that sparse computation is theme that will be revisited during the hardware optimization part of this work. One can deliberately insert zeros during training while keeping the hardware architecture in mind. In this way, since zeros were allowed in specific parts and not in others, the optimization is also done in a hardware level.

The methods targeting parameter reduction are usually followed by a fine-tuning

phase in order to minimize the effect on the accuracy [2].

Sparsity exploitation means to take advantage of the intrinsic redundancy in data representation. This aspect of neural networks has been explored by some works. Proposed by [45], Stitch-X is a DNN inference accelerator that by combining spatial and temporal reduction balances dataflow complexity in face of sparsity. It utilizes a Parallelism Discovery Unit (PDU) that stitches together the input activation and weight pairs for producing reducible partial sums.

Similarly, the accelerator proposed by [84], Cambricon-X, also aims to exploiting sparsity and irregularity of NN models while also using 16-bit fixed-point representation. Related approaches can also be observed from [61, 38, 39].

3.7 Algorithmic Optimizations

In order to reduce complexity, some operations can be transformed, and algorithmic optimizations applied.

When concerned about CNNs, a common approach is to instead of computing complex convolutions in the time-domain, choosing to simply calculate multiplications in the frequency-domain with Fast Fourier Transform (FFT) [81, 2]. If a more hardware-friendly manner is required then Winograd transformation can be applied [48, 58, 48, 70].

At this point, an important highlight must be made. From equations 3.1, 3.3 and 3.4, it is obvious why matrix multiplication is crucial for the computation of neural networks and the importance in optimizing these operations.

The GEMM transformation basic idea is to map convolutional and fully connected layers as General Matrix Multiplications. In the simplest format, GEMM computes the operations given in equation 3.5, where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are matrices, α and β are scalars and op(.) denotes either the original or transposed matrix [25]:

$$C = \alpha \text{op}(\mathbf{A})\text{op}(\mathbf{B}) + \beta\mathbf{C} \tag{3.5}$$

Previously, it was mentioned that the biggest portion of the weights are used by fully connected layers. This is an important fact when using GEMM implementations for computing these multiplications because batch processing can be used.

In batch processing, multiple inputs are provided instead of one, Figure 3.3 depicts

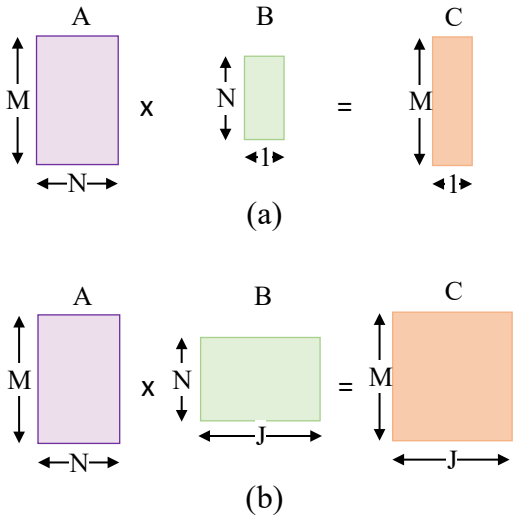

Figure 3.3 *(a) Matrix-vector multiplication - Level 2. (b) Matrix-matrix Multiplication - Level 3. Adapted from [70].*

this case in (b), if the inputs are the combination of vector B in (a). The throughput can be improved while memory bandwidth is maintained when instead of loading weights multiple times, they are loaded once per batch. [2].

Indeed, from Basic Linear Algebra Routines (BLAS) three canonical computation models can be performed: vector-only operations, matrix-vector and matrix-matrix operations. Note that they correspond to levels, respectively Level 1, Level 2 and Level 3. Figure 3.3 shows matrix-vector in (a) and matrix-matrix operations in (b).

The lowest level can be used to implement the other two and so forth. Each of these levels can be mapped for specific usage. On one hand, Level 3 operations are highly desirable for dense matrix-matrix calculations and perform well for batch mode, on the other hand, Level 2 is a good fit for batch-1 implementation [24].

So far, only software optimizations were discussed. Although these optimizations were placed under software, they will directly impact on the hardware used to implement the computations.

4. INFERENCE ACCELERATORS

This chapter reviews hardware-based acceleration techniques and proposals for neural networks inference. As such, important aspects for a hardware efficient design and system level architecture will be discussed.

The list of works targeting deep neural networks inference accelerators is extensive. Although this is not a particularly new field of research, the first neural network FPGA implementations are dated back to 1990's [44], there was an explosion of works recently, as can be seen in [72, 7, 31, 85] and others.

However, the reviews showed in this work concentrate mainly in efforts proposed from 2014 to the present-day for three reasons. Firstly, the number of works in this field is huge, secondly, NNs have become deeper after 2014 which changed their computational requirements and thirdly, as mentioned earlier this work is not meant as a survey.

4.1 Hardware Efficient Design

Recently there was a shift in the main purpose of the design of DNNs. Surely, in the early days the main objective was to achieve the maximum accuracy. While this is still true, the impact of the design in the hardware implementation is gaining more and more importance. In this sense, the codesign of DNN models and hardware can be classified as an effort for maximizing accuracy and throughput, while minimizing energy and cost. [70].

FPGAs provide a high level of flexibility for hardware implementation. However, there are at least two big challenges in FPGA based accelerators [30]:

- the current working frequency of FPGA is usually in the range of 100MHz to 300MHz, much less than general purpose architectures,
- the abstraction level for implementing neural networks on FPGAs is much lower, making it a much more difficult task.

In order to address these challenges, there are some trends in the FPGA industry to look at. The operating frequency of usual designs should have a big improvement with new technologies, as is the case of Intel's HyperFlex. Additionally, the on-chip memory and the off-chip bandwidth should increase considerably, the latter with the use of HBM (High Bandwidth Memory) technologies [57].

Regarding the second challenge, the software ecosystem for FPGAs is becoming more mature. The biggest FPGA's industry players, Intel and Xilinx, have been supporting the use of High-Level Synthesis (HLS) tools which offers the possibility of using high level abstraction languages for programming FPGAs. This support, brings the advantages of these devices to the reach of more people than only hardware experts [57].

It is important to highlight that scalability is the biggest issue when looking forward on FPGAs and deep learning. In order to achieve successful implementations, they must scale in data sizes and architectures, since the research in deep learning is still on-going and the pace in which new models and techniques are being developed is very high [44]. One may refer to this as exactly the lead which FPGAs represent.

4.1.1 Parallelism Exploitation

General Purpose Processors mostly employ a temporal architecture for parallelizing computations, in the form of Single Instruction Multiple Data (SIMD) or Single Instruction Multiple Thread (SIMT) techniques, for example. In contrast, FPGA-based designs are usually constructed on top of spatial architecture for dataflow processing. The main difference between these two architectures is the data passing format. In the first, data can only be fetched from memory hierarchy and the compute element cannot communicate in a direct manner with another. In the second, data is passed from one unit to the other directly. [70].

Surely, this aspect reflects directly into DNNs efficient design. In this context, data-path optimizations can be adopted to address the problem of efficiently using FPGAs for inference accelerators.

The usage of systolic arrays is well-known for this purpose. These are grid structures, usually arranged as depicted in Figure 4.1, that are formed by several processing elements (PEs). State-of-the art implementations employ a limited number of these units on the FPGA, each of these units can be reused by iterating data through them [2]. The utilization of systolic array architecture for CNNs in and end-to-end automation flow is demonstrated by [73].

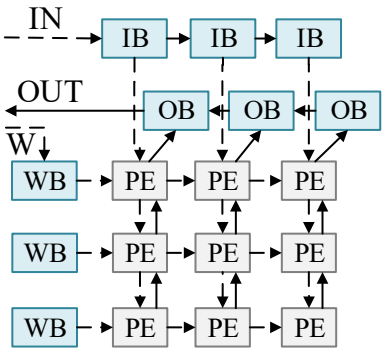

Figure 4.1 *Systolic array architecture. Adapted from [73].*

It is important to formalize the possible sources of parallelism in DNNs in order to understand the ways of exploring it. In this context, at least two forms can be readily identified, batch parallelism and inter-layer parallelism. The former was already mentioned when discussing about matrix-matrix multiplications, but it means to serve a group of inputs with the objective of reusing data and decreasing external memory accesses. The latter refers to the scheme in which the computation is launched in a pipelined fashion. [2].

Observe that these sources of parallelism are exactly aligned with the extraction of maximum performance from a FPGA. In fact, the industry claim is to have a peak performance of over 1 TFLOP/s for the DSP (Digital Signal Processor) blocks in the FPGA. However, the task of fully pipelining and loop unrolling for maximum parallelization is not as easy as it seems. [73].

Along these lines, loop unrolling is a key technique in hardware optimization. The idea of unrolling loops in an FPGA is basically a trivial one, the downside is the trade-off between performance and resource utilization. An important side note though is that if poorly chosen the unrolling parameter can cause severe hardware underutilization. This is particularly important since different layers have very diverse loop dimension. [30].

There are many methods proposed in literature for choosing an optimal value for loop unrolling factors. The challenge is to derive a parameter that at the same time minimizes the memory access and maximize resource utilization [2].

4.1.2 Resource Utilization

As mentioned previously in Section 4.1, although being improved recently, the on-chip memory capacity on FPGAs is still small for deep designs. This means that

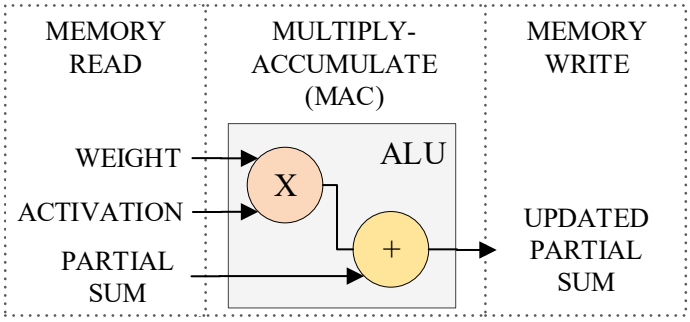

Figure 4.2 *Memory access in one MAC operation. Adapted from [70].*

off-chip memory must be used [2]. Since this is inevitable, a caching memory hierarchy should be implemented, it is usual to have a two-level cache in FPGA-based implementations. One may question the need of such schemes, for that purpose Figure 4.2 shows the need of three memory read operations and one write per multiply-accumulate (MAC).

In this fashion, the use of a caching system is simply an exploitation of the spatial architecture provided in FPGA implementations. The other option is to utilize off-chip memories and in the case of DRAMs (Dynamic Random-Access Memory) take much more energy to access the memory than the computation itself [70].

From the same perspective, data reuse plays an important role in this scenario. Even if DRAM accesses are needed, since they are so expensive, the fetched data should be reused as much as possible. For a comprehensive explanation on data reuse schemes on dataflows, refer to [70].

To reduce off-chip memory bandwidth requirements and minimize data movement, fused-layer accelerators can be used, as first demonstrated by [3]. This technique can be combined with other optimizations, for example [85] offers the use of Winograd in its convolution blocks templates with the addition of layer fusion optimization.

It is also important to mention general FPGA-based implementation optimizations. Whenever a design is devised the target is to fully utilize the FPGA capabilities. In this sense, many important guidelines must be followed. Among those, two are absolutely important, the usage of DSP blocks and the improvement of working frequency.

In the case of DSPs, the adopted bit-width is crucial. This is because, depending on the vendor and on the FPGA, this can vary and hardened portions on FPGA in general achieves higher frequency and consequently performance. For example,

the accelerator demonstrated by [31] utilizes the 8-bit fixed-point representation for packaging two operations of 8x8 bits into one DSP of the FPGA. The system presented in [64] applies dynamic-precision data quantization for VGG16 model by using an automatic flow, a small accuracy loss is introduced with the model under 8/4 bit dynamic-precision quantization.

Recently, a trend in FPGA industry is to support floating point operations natively, as is the case of Intel's Stratix10 device, offering up to 9.2 TFLOP/s of 32-bit floating point performance [57].

4.2 System Architecture

From a system level perspective, it is possible to identify some trends in neural networks implemented in FPGAs.

When focusing in HDL (Hardware Description Language) model-based approach, the main idea is to automate the process of generating the HDL description taking into account the selected network. This means that the generated hardware is fine-tuned for a determined neural network and the best performance can be achieved for that particular hardware [30].

Instruction based methods, on the other hand, do not modify the underlying hardware, thus several neural networks can run on the same hardware implementation. An application that needs neural network switching would target this implementation, since the change can be done in real-time [30].

Finally, these two methods can be combined into a solution that besides optimizing the hardware, also uses a set of instructions compiled corresponding to the network description. [30].

4.2.1 Hardware

A neural network inference accelerator is typically formed by the parts showed in Figure 4.3. In a high level overview, the host CPU plays the role of a scheduler in which it will issue commands to the logic and monitors its status until the end of the computation is reached. For controlling the operation on the FPGA, a controller must be implemented, it can either be a finite state machine or an instruction decoder [30]. Some implementations can use a soft-core processor synthesized in the FPGA for this purpose, as is the case in [24].

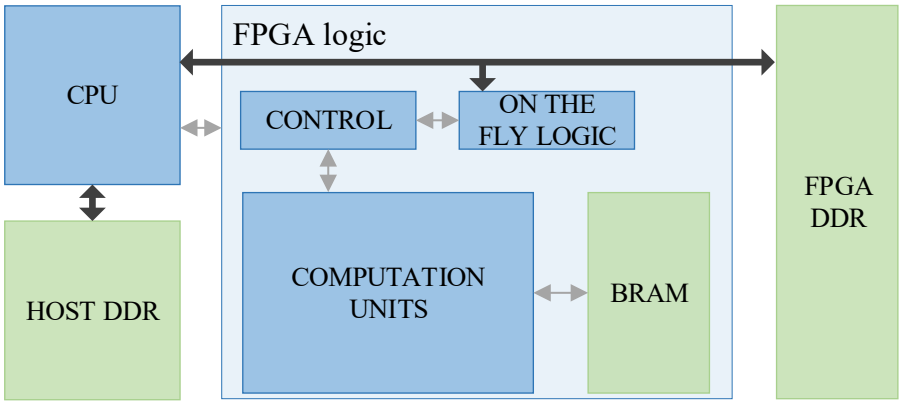

Figure 4.3 *Block diagram of a typical FPGA-based inference accelerator implementation. Adapted from [30].*

From the memory viewpoint, for the reasons already referred previously, for a large model an external memory is required for holding all the needed parameters. In such cases, a memory hierarchy scheme must be devised, as for example using the external memory and the on-chip memory as cache.

Obviously, this generic system can have several modifications and optimizations depending on the main objective, its requirements. For example, in a simpler solution only one memory could be used in the entire system, which can be seen on the small configuration version of NVDLA [58]. For more advanced and high performing solutions, besides the two external memories used, a micro-controller could be placed between the CPU and the FPGA, so the host is freed from handling all the interruptions in the system as is the case in the larger configuration for [58], which could even be implemented as a soft-core processor on the FPGA.

The content of the computation units can vary abundantly, as discussed previously systolic arrays can be used to form convolutional compute engines. Inevitably, logic for the non-linearity computation and algorithmic optimizations can be added to overall structure of the FPGA. Again, an example is the NVDLA [58].

4.2.2 Software

From the software perspective, the execution model of an inference accelerator can be compared to those of GPP architecture. That is because it usually involves two phases, the compilation and execution, as depicted in Figure 4.4.

Analogously to the translation of a program targeting a specific architecture i.e. x86, the compilation step targets a DNN accelerator, in which case the neural network

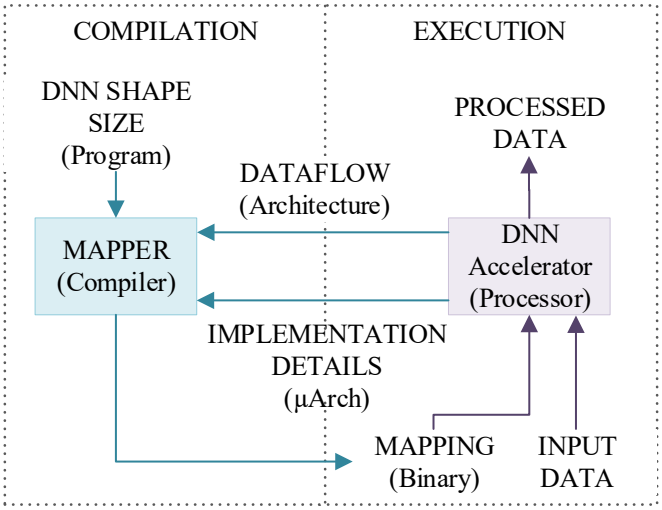

Figure 4.4 *Execution model analogy between DNN inference accelerators and GPP. Adapted from [70].*

is translated into a hardware-compatible computation mapping [70]. Note that in this phase, or even before it, several neural network optimization techniques (as described in Sections 3.6 and 3.7) can and should be applied to deliver the best hardware friendly version possible.

Observe from the diagram in 4.4 that the mapper, in the compilation phase, needs as input the hardware implementation details. This is a crucial step in producing high-performing accelerators.

Additionally, it is important to notice that commonly an intermediary representation is needed to pass the model of the neural network to the accelerator. This representation is usually very different from the original model, for example, it can have a fused layers implementation, as proposed by [3]. The objective of this method is to reduce the use of external memory usage and as such improve the performance of the design.

To address the same problem, it is also possible to use a persistent approach, in order to pin the models in the on-chip memory of the FPGAs. This approach must be supported by an entire infrastructure at scale, since if only the on-chip memory is used, large models cannot fit in one device but can fit across several FPGAs. Microsoft's project Brainwave targets exactly that.

4.3 Tools and Architectures

One of the main concerns in implementing efficient inference accelerators in FPGAs is the heavy and difficult task of doing so. In this context, to ease this process, many works provide frameworks for automatically generating the underlying hardware description. This is the case of [72], further extended in [7]. FINN is a framework targeting Binarized Neural Networks inference accelerators on FPGAs, a custom architecture is built specifically for a given topology. The accelerator generation process uses two inputs, the trained BNN and an FPS target for the final implementation.

Also relying on data quantization technique allied with a design flow proposition, Angel-Eye presented by [29] targets the mapping of CNN onto embedded FPGAs. The input to the flow is CNN model from Caffe that is quantized and fine-tuned to increase accuracy, in the next step an instruction sequence is devised through a compiler for the execution of the model. The underlying hardware implementation is parameterized and runtime configurable.

In a similar manner, the framework proposed by [85] addresses the generation of efficient CNN models for domain-specific applications on FPGA. The key component on this framework is the hardware design template used with Winograd optimized convolution blocks and fused-layers approach. Transfer learning is used for fine-tuning the trained input model for a given application. In this sense, the inputs for this design flow are the domain knowledge, pre-trained models, the platform specifications and the requirements. As a result, the optimized model design is generated.

Another approach for abstracting these difficulties, is the use of Instruction Set Architecture. Cambricon, proposed by [47] is an instruction set specific for neural networks accelerators, it comprises the usage of a load-store architecture with 64-bit instructions devised from the study of NN techniques. Indeed, no vector register file is used to supporting common computations on NN, so the data is kept in on-chip memory being visible from the programmer perspective.

The combination of an Instruction Set with a framework targets an even bigger scope. These solutions offer the possibility of completely disengaging programmers from hardware design. The work presented by [68] has as inputs a pair of (DNN, FPGA), in which the DNN model is specified by a high-level programming interface, i.e. similar to Caffe description, and is further translated into a macro dataflow Instruction Set Architecture (ISA). Similar approaches are proposed by [28, 49, 28].

Another important aspect is the system solution in which these inference accelerators are deployed.

The approaches described so far usually present high flexibility of implementation, either regarding the DNN models supported or the target hardware platform. The work proposed by [52], NEURAghe, is aimed at Zynq SoCs (System on a Chip) and investigates the usage of the ARM cores with a convolution specific processor deployed on the reconfigurable logic. The ARM host program is automatically generated for the execution of the fully-connected layers and data marshaling.

This is one of the several examples targeted to SoCs solutions focusing on embedded architectures for NN acceleration. One can also mention [29] and [39] as recent approaches in this context.

In this sense, when one proposes an FPGA-Based inference accelerator it is usually designed to be a co-processor. One can either use SoCs that integrates multi-core processors, memory interfaces and reconfigurable logic on the same board, for an embedded approach or target the usage of the pair host and FPGA. This last approach utilizes PCIe I/O bus for communication between the host and the FPGA device, as is the case in [28]. Observe that these two types of system implementation can be interchangeable by some modifications.

The important point here is that all these implementations are usually targeted for real-time mobile applications, that are performed on the device. From the server point of view, [29] arguments that aiming at a specific network implementation is a good choice for achieving extreme hardware performance, but a problem when targeting real-time mobile applications that run on the device. This work agrees with this sentence, a specific neural network is indeed implemented for extreme performance purposes targeting a cloud environment structure.

The oldest work cited here is DaDianNao [14] and its correlation to the present work is symbolic. Motivated by the memory storage and bandwidth requirements in DNNs, as well as its correspondent limitations, the authors propose the usage of a multi-chip system, in which each on-chip storage could be used for implementing the biggest DNNs architectures. The proposed architecture is named a supercomputer for its capacity. Note that the underlying idea of scaling-up Deep Neural Networks across several chips is basically the core of a cloud computing environment.

4.4 Inference Accelerators in Cloud Environment

From the inference accelerators perspective, they are usually designed as embedded applications targeting real-time solutions, which is the interest of academia, as discussed in the previous paragraphs. When it comes for a cloud environment, two examples will be exposed here.

Google's TPU [37] is a custom ASIC designed to be a coprocessor on the PCIe I/O bus for easier integration with existing servers. The main objective was to run inference models to reduce the host CPU interactions in this process, also to present flexibility for several and evolving DNNs. The core of this accelerator is a Matrix Multiply Unit, fitting the highest level of canonical computation model, Level 3, as described in Sections 3.6 and 3.7. The instructions are sent from the host and allocated in an instruction buffer. The NN is compiled from TensorFlow to an API that can run either on GPUs or TPUs. The TPUs were deployed since 2015 in data centers for inference acceleration and can use big batch sizes for improving performance.

From another perspective, Microsoft's Brainwave project [17] [24] targets the use of FPGAs in their Azure cloud. In this sense, the acceleration scope is larger, since one of the applications is indeed DNNs inference. Their goal is offering the possibility of running real-time applications, in this sense, latency is the most important figure and batch size 1 is used for this purpose. The Neural Processing Unit (NPU) targets a different approach for low-latency solution, as demonstrated in [69], in which the model parameters are pinned in on-chip memories. It presents an ISA for accommodating a wide range of DNN models and to ease the task of its programming. Brainwave is built on top of the Catapult enhanced-servers [10], in which the FPGAs are PCIe attached to dual CPUs and are physically in-line between the server Network Interface Card (NIC) and the switch enabling point-to-point connectivity, using RDMA-like (Remote Direct Memory Access) protocol, between thousands of FPGAs in the data center. The tool flow is responsible in accepting the DNN model and mapping it to the distributed system architecture utilizing the hardware microservices in which the NPU is synthesized in the targeted FPGAs.

The approach aiming at low-latency is well funded, as being referred extensively in this work. In Google's TPU paper [37] when discussing their deployment, the conclusion regarding NN inference applications in data centers is that they were surprised by the strong response-time required by some applications in which the preference was shifted from bigger waiting times with bigger batches to reduced latency in inference.

Still in this section, it is important to mention a few words on the role that industry is playing in this field of research. From all the implementations and studies cited here, at least one in three is directly mapped to the biggest industry players.

As is the case for FPGAs, Xilinx in [72] and [7] heavily and consistently work with quantization, making a heavy effort in proving tools for abstracting the difficulties in implementing efficient inference accelerators in their FPGAs. Their latest work [8] evaluates the FINN framework for different data types in an AWS (Amazon Web Services) instance, since Amazon offers FaaS, FPGAs as a Service in their public cloud. Just recently, Xilinx revealed Versal ACAP (Adaptive Compute Acceleration Platform) which combines programmable logic with a set of new features including Intelligent Engines (Software programmable vector processors), refer to [79] for the Versal architecture overview. Notice that the target of these devices is a wide range of applications, including cloud, network and embedded.

Intel on the other hand, review similar optimization approaches for using with their new FPGAs family [57], Stratix 10, which in a partnership with Microsoft is used in large scale cloud environment with project Brainwave. NVIDIA undoubtedly the biggest player in the GPUs market, has open-sourced their effort in devising a complete software/hardware configurable solution in NVDLA [58] and also proposes a sparsity accelerator, which is not a good fit for GPUs, in Stitch-X [45].

Google and Microsoft have been investing heavily in their cloud infrastructure from two different approaches. Google's TPU [37] is an ASIC based DNN inference accelerator whilst Microsoft recently deployed its Brainwave project [17] in FPGA based inference accelerators.

Undoubtedly, Microsoft's Brainwave project is the most inspiring work, regarding cloud architecture, for the present implementation. The work presented here also aims at ultra-low latency applications, but uses a different approach than the one devising a NPU. As already cited, it is possible to defend specific implementations when targeting extreme high performance. This is the case of the implemented neural network, although it presents a synthesis time parametrization, it is aimed only at MLPs. Another important mention is the system level architecture, in which, by using Software Defined Networks in the reconfigurable logic, it is possible to route packets between FPGAs and achieve the same result as not having software in the loop. Since, the MLP implemented does not need any instruction or runtime configuration it can work in a stream-like manner achieving the intended ultra-low latency requirements.

5. METHODOLOGY

This work presents a comparison between multiple implementations for the acceleration of an anomaly detection neural network. The neural network model creation and training is not a part of this thesis. The original model was quantized, as an optimization step for a hardware efficient implementation. This quantization is also out of the scope of this thesis work. As such, the comparison is done between:

1. the original model executed in CPU utilizing Keras,
2. the original model executed in GPU utilizing Keras,
3. the quantized model executed in FPGA utilizing GEMX,
4. the quantized model executed in FPGA utilizing SDAccel,
5. the quantized model executed in FPGA utilizing CRUN.

The original anomaly detection neural network model was implemented and trained using Keras framework [16]. Thus, the original model could be executed in CPU or GPU. The other three implementations utilized a quantized model with different approaches, GEMX is a high-level implementation, whilst SDAccel and CRUN utilizes a hardware RTL (Register Transfer Level) kernel. For all cases the pre-processing of the data was not included in the latency measurements.

The steps of execution of each experiment are described in the next two sections. The first introduces all the experiments, referred as reference implementations, with exception of CRUN that is specifically introduced in the subsequent section. Only CRUN and SDAccel were implemented from scratch and CPU, GPU and GEMX were basically run for comparison purposes. The validation process is also described in this Chapter.

5.1 Reference Implementations

The trials introduced in this section were performed for comparison purposes with the main implementation of this thesis, the CRUN. This means that most of these

cases were conducted from ready-made implementations with needed modifications.

From the order of the implementations presented here, it is also possible to outline the order in which the experiments were made. For example, CPU was the first implementation and SDAccel the last. They represent steps for the final solution constituted by CRUN.

5.1.1 CPU & GPU

The original anomaly detection neural network model was implemented and trained using Keras framework [16]. Note that model development and training are not in the scope of this work. However, for a simple comparison with the inference metrics of the original model, in CPU and GPU, Keras was used.

Keras, written in Python, is a high-level neural networks API capable of executing either on CPU or GPU by running on top of TensorFlow™ [16]. TensorFlow™ is an open source software library based on data flow graphs for numerical computation [71].

The server used for both cases was Intel® Xeon® Gold 6130 CPU @ 2.10GHz [34] and the NVIDIA® Tesla® V100 Data Center GPU was employed [59] for GPU measurements.

In this scope, ready-made Python scripts with the original Keras model were used and modified for predicting the model and timing its computation. Only one use case was performed here, measuring the time for predicting one or more inputs of the model using the predict method in Keras. This was performed for CPU and GPU independently. Also, different batch sizes were used for analyzing the relation of batch size and latency. The pre-processing of the data is not included in the latency measurements, but only the inference latency.

5.1.2 Xilinx GEMX

The original anomaly detection model from Keras was quantized for the implementation with GEMX from Xilinx [74]. The quantization and GEMX implementation details are not part of this work.

GEMX is a General Matrix Operation library used for acceleration of BLAS-like matrix operations. It is used on SDAccel supported FPGA cards from Xilinx, which

comprises Xilinx KCU1500 and Xilinx VCU1525. In this work, this implementation was carried out in VCU1525 Reconfigurable Acceleration Platform featuring Xilinx® Virtex® Ultrascale+™ FPGA [78]. The host was the same as in the CPU implementation.

GEMX library is composed of three components: an engine library, a host code compiler and an application or system building environment. The engine library offers blocks for building matrix operation accelerators on FPGAs, it comprises a set of C++ templates that can only be used on SDAccel supported platforms. The compilation of the host code is performed by the host code compiler that translates the matrix function calls into a sequence of instructions for computing matrix operations on FPGAs. Finally, the building environment uses GNU make flow for generation of the host code and FPGA's image [74].

A GEMX Python API was used for sending the data to be predicted to the FPGA. Only one test case was performed with GEMX utilizing one batch size, which was measuring the time spent for the FPGA computing the needed calculations and returning the predictions. The pre-processing of the data is also not included in the inference latency.

5.1.3 Xilinx SDAccel

An RTL kernel of the anomaly detection neural network was developed for and implemented utilizing Xilinx SDAccel™ Environment [77].

SDAccel™ is a framework that offers the possibility of developing and delivering accelerated data center applications on FPGAs. It uses standard programming interfaces, making it easy to use for developers of accelerated applications with no prior knowledge about hardware design. However, the tool flow allows the usage of hardware-centric approaches for development of the accelerated kernel [77].

This environment targets acceleration hardware platforms as Virtex® Ultrascale+™ FPGA VCU1525, which was used for carrying out this implementation. The host was the same as in the CPU, GPU and GEMX implementations.

The SDAccel™ offers support for kernels developed in OpenCL™ C, C/C++ and RTL [77]. In this thesis work, a RTL kernel was used and some requirements for its development were fulfilled.

From the software side, the RTL kernel was designed for being started when called by the host, compute all data values and return it at the end of the operation.

From the hardware side, AXI4-Lite interface slave is used for accessing the kernel control registers and AXI4 master interface for communication with the memory, for sending and returning data.

For SDAccel, the test case was to measure the latency between sending the data to the RTL kernel on FPGA and obtaining the results for that data using C code. Two batch sizes were used for accessing the effect of the batch size in the latency measurements. Observe that the pre-processing of the data is not included in the measurements.

5.2 CRUN Implementation

CRUN framework is part of an in-house infrastructure, which is composed by software and hardware components. This thesis utilizes its hardware components, here referred as CRUN shell, for integrating with the RTL kernel of the anomaly detection neural network. More information on CRUN framework can be found in [41].

The laboratory setup showed in Figure 5.1 (a) and (b) contains two servers with dual socket Intel® Xeon® E5-2680 v4 @ 2.40GHz CPUs of 64-bit and x86 architecture, the CPUs have 14 physical cores each and hyperthreading enabled, providing 56 treads in total and 128 GB of DDR4 memory. For network communication the NICs (Network Interface Controllers) used are Intel® 82599ES 10 Gigabit Ethernet Controller. Additionally, the switch connecting NICs and FPGAs referred in Figure 5.1 is model QFX5100-48S from Juniper® Networks.

The implementation was carried out on KCU1500 data center board with the Xilinx® Kintex® Ultrascale™ FPGA. For more information on the board, refer to [76].

Vivado Design Suite [80] was used for developing, verifying, validating and integrating the RTL computation kernel for anomaly detection neural network and CRUN shell.

The customizable Integrated Logic Analyzer (ILA) [75] IP core was also used for monitoring the internal signals of the design. The latency measurements for the anomaly detection RTL kernel were first analyzed from simulations and posteriorly confirmed by the usage of the ILA core. Then, by using IXIA board NOVUS-R100GE8Q28 the CRUN shell only latency was obtained. This setup was part of the Hardware (HW) only measurements, depicted on Figure 5.1(b).

For obtaining latencies from the Software (SW) level, another method was used. The anomaly detection data was sent through the network over Ethernet to the

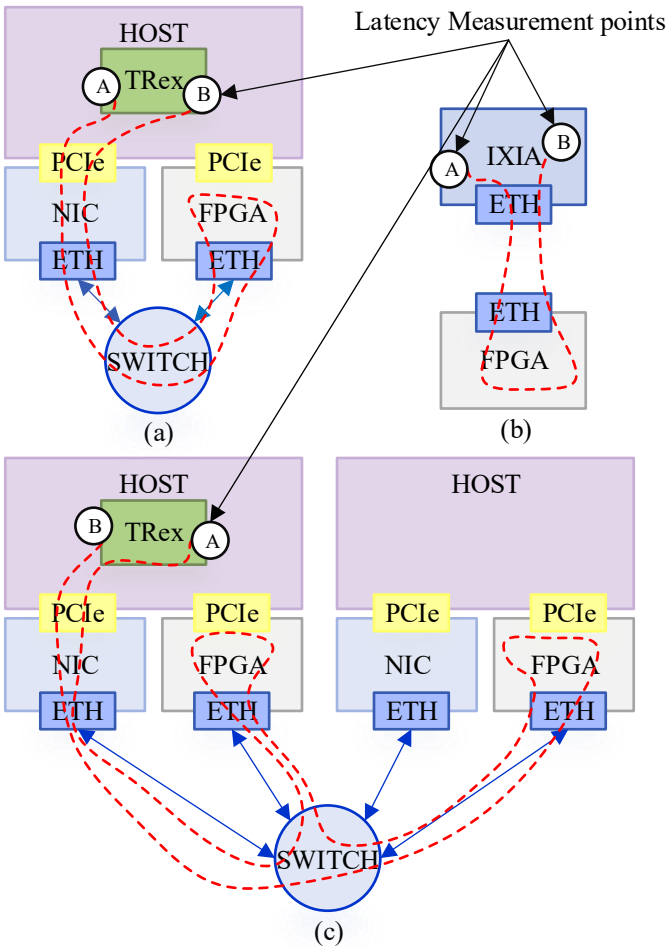

Figure 5.1 *CRUN Test Cases Lab Setup. (a) represents the configuration using one FPGA. (b) shows the usage of IXIA board for HW measurements. (c) represents the distributed version of the system, using two FPGAs.*

FPGA, TRex was used as application from host OS. TRex [19] is an open source traffic generation tool, that runs on standard Intel processors. It uses DPDK (Data Plane Development Kit) and supports stateful and stateless traffic generation modes. Both modes were used for the latency measurements discussed in Chapter 7. The use cases utilizing TRex are depicted in Figure 5.1(a) and (c).

There is the possibility of running TRex on a hypervisor with virtual NICs. In this work, however, bare metal was employed.

Figure 5.1 also shows the latency measurement points for each setup. For example, in (a) the round-trip latency is measured from Trex following point A to point B.

The payload of each packet sent through TRex was generated based on the anomaly detection data for prediction. The Ethernet packet fields were configured for the correct destination, depending on the lab setup utilized. For example, in the case

of Figure 5.1(a) the host was the destination, but in the distributed case depicted in (c) the first FPGA should send the data to the second FPGA, so the destination was changed.

In summary, three experiments for latency measurement were carried out with CRUN framework and the anomaly detection RTL kernel: from SW perspective utilizing one FPGA, from SW perspective with the distribution of workload to two FPGAs and HW only measurement without SW layer in the loop. For all the experiments with CRUN the pre-processing of the data is not included in the latency measurements, but only the time between sending the data and receiving it.

5.3 Validation

The CPU and GPU implementations refer to the usage of the original Keras model without quantization. Hence, they correspond to the base implementation for this acceleration.

GEMX was the first implementation that utilized a quantized model, as such, accuracy study was carried out for comparing CPU and GEMX results. Note that this was done previous to this thesis work.

However, the validation process carried out for the experiments of this thesis were fully executed for Xilinx SDAccel and CRUN implementation. They were both validated against GEMX implementation, which constitutes the first quantized experiment. Validation scripts utilizing Python were especially developed for this purpose and used throughout validation process.

Firstly, the RTL anomaly detection neural network was verified utilizing its own test bench written in System Verilog in Vivado environment. Once the behavior was verified, it was ported for SDAccel environment and the hardware emulation flow was used for testing the software integration. Finally, the system was fully built and tested on hardware.

For CRUN implementation, the validation was also supported by Python scripts for creating the necessary Ethernet packets with the correct payload data. Thus, several pairs of input and expected output were validated for this system.

6. IMPLEMENTATION

At first, this Chapter describes the CRUN framework, an in-house development. The anomaly detection neural network is then presented and explained in a high-level overview. In the sequence, its hardware implementation is discussed along with optimization methods utilized. The MLP, for anomaly detection application, presented here is just one example from many candidates. Observe that it could be any other cited on the subsection 2.3. In this case, the study was done in a general format, which means that the work was realized as if the neural network had different characteristics, an example is the distribution of the workload to two FPGAs.

6.1 CRUN Architecture

CRUN is a framework composed of software and hardware components. Since it is still under development and it is an in-house effort for enabling the acceleration of different applications in a data center environment, its architecture will be presented. However, only the hardware components will be discussed because the software is out of the scope.

The CRUN ambit is bigger than what was used here, thus only a brief overview of the most important components will be discussed in a top-down manner.

Figure 6.1 shows the high-level view of a distributed system enabled by CRUN, composed of three example servers. The switch connects the local network composed of hosts and FPGAs to the Internet.

The content of the three hosts are identical. Each server is connected to its NIC and FPGA daughtercard through PCIe. In CRUN a Virtual Machine (VM) is used for the deployment of the applications that will be accelerated. However, in this work the application (TRex) runs on bare metal.

The framework also supports the software layer. Figure 6.1 shows two important software components: BRO server and BRO client. They are responsible for all

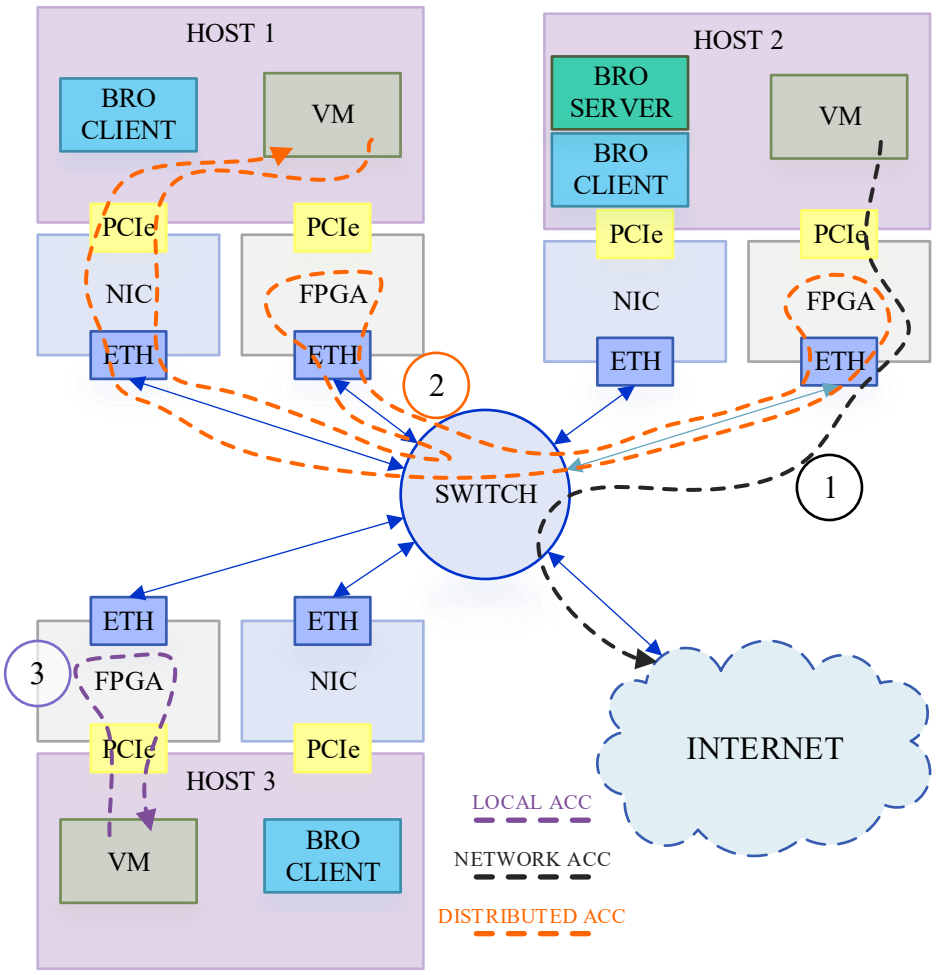

Figure 6.1 *CRUN architecture overview. VM represents the Virtual Machine present in each host. Distributed acceleration is represented by red dashed line and is the only scenario utilized for accelerating anomaly detection neural network.*

the operations comprehending the deployment and management of the VMs and applications through the hypervisor, the programming of the FPGA bit file, the configuration of the network and the application life cycle.

Three types of accelerators are supported by this system, marked in Figure 6.1 by numbered routes. Path 1 from host 2 (black dashed line) shows the network acceleration mode, in which application data is in-line accelerated before going out of the data center. Path 2 from host 1 (orange dashed line) introduces the distributed acceleration case, where data is routed to a chain of two FPGAs. Path 3 from host 3 (purple dashed line) shows the local acceleration, in this mode data is transferred through PCIe to the accelerator and back using DMA (Direct Memory Access).

However, only one of these acceleration modes is used in this thesis, the distributed acceleration in two distinct ways. One is exactly as the picture shows, with two FPGAs in which data is transferred to the FPGAs through the network. The other is by using only one FPGA.

A closer look at the FPGA architecture of the system is given in Figure 6.2. All the components on Figure 6.2 with exception of the Accelerator Hardware Unit (AHU) are a static part of the reconfigurable logic and are referred to as CRUN shell.

The control elements of the system are represented in yellow, they can be accessed through a PCIe driver that performs memory map access to the shell and AHU by different address spaces. The green blocks represent the local acceleration mode of Figure 6.1, in which data is transferred through PCIe via DMA directly to the accelerator and back.

The blue blocks of Figure 6.2 represents the Ethernet stream path and are the most relevant part for this thesis work. Observe that Ethernet frame packets are used for transferring the acceleration data, refer to [1] for more information on the composition of such packets. P4 RX and P4 TX are SDN components and provide the networking functionality of the shell.

ETH RX is responsible for checking and translating the incoming physical Ethernet packet into an equivalent AXI4-Stream packet, no application specific information is processed from Ethernet fields. P4 RX main task is to find from the headers field the correct AHU destination for sending the income packet, it also removes the headers for delivering only payload data for the accelerator.

The ROUT RX component receives the packet payload and routes it to the corresponding AHU. In the case that AHU is not ready the packet is dropped. This action is taken because the stream path function on line rate, and it is responsibility of the AHU to support the correct throughput for the application.

Immediately after the AHU computation, ARB TX is responsible for receiving the packets generated from the AHU and deciding which should be served. In the sequence, P4 TX task is to build the IP headers for the specific payload it received from AHU. The configuration of addresses and ports, which constitutes the constant fields in the headers, is updated during runtime by PCIe control port.

The last step on the stream path is ETH TX that converts AXI4-Stream to physical Ethernet packets for being transferred to the data center network. Once on the network, the packet will be routed to its destination.

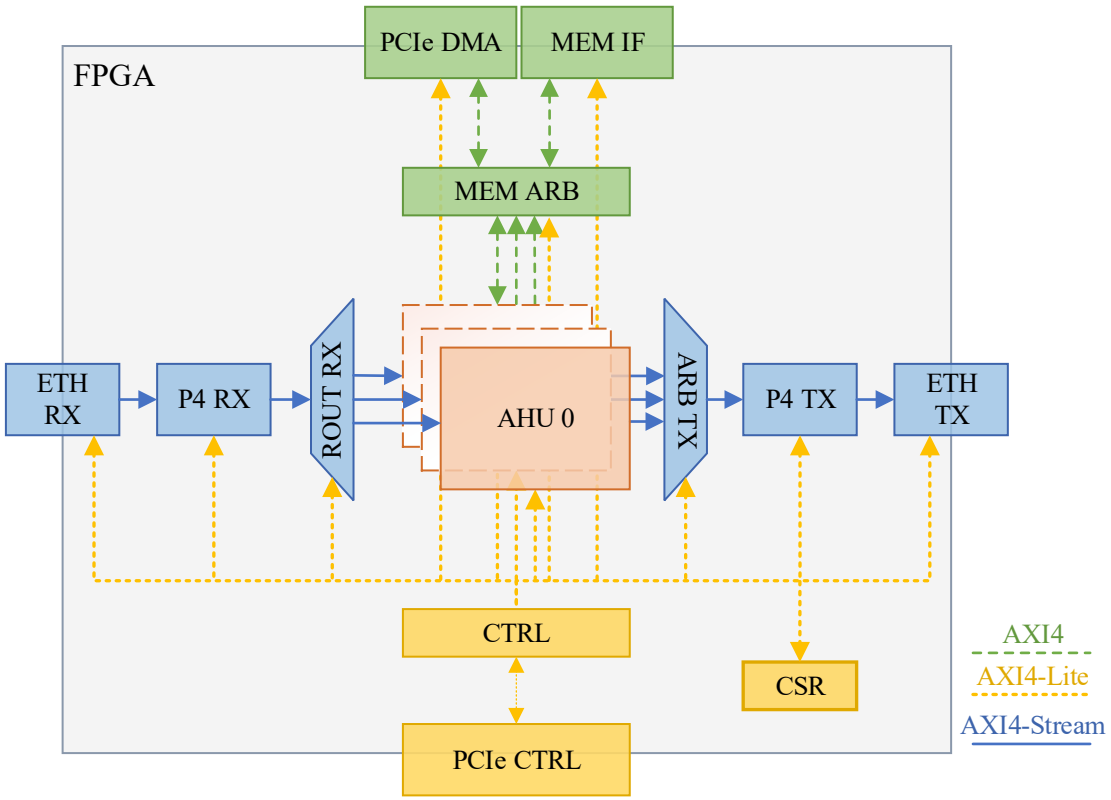

Figure 6.2 *CRUN FPGA architecture overview. The stream data path is represented by the blue components and is the only one used in the acceleration of the anomaly detection neural network.*

Note that in this thesis work the AHU corresponds to the anomaly detection MLP discussed on next section. One can define AHU as the hardware element that processes the data, while the CRUN shell objective is to serve the correct data for the specific AHU.

Here, the standard interface of AHU is presented on Figure 6.3 (a), different interfaces are provided for control and memory access.

The usage of the CRUN framework in this work is limited and as such its capabilities are beyond its scope. Anomaly Detection neural network AHU implementation does not utilize any control or direct memory access scheme (either for host or on-board FPGA memory), but solely the RX and TX using AXI4-Stream.

Figure 6.3 (b) shows the AHU from the shell point of view. MLP LAYERS is the RTL kernel with the anomaly detection neural network, detailed in Figure 6.6.

AXI4-Stream is employed for interfacing with CRUN shell. Note also that FIFOs are inserted between MLP LAYERS and the AXI stream interface for supporting

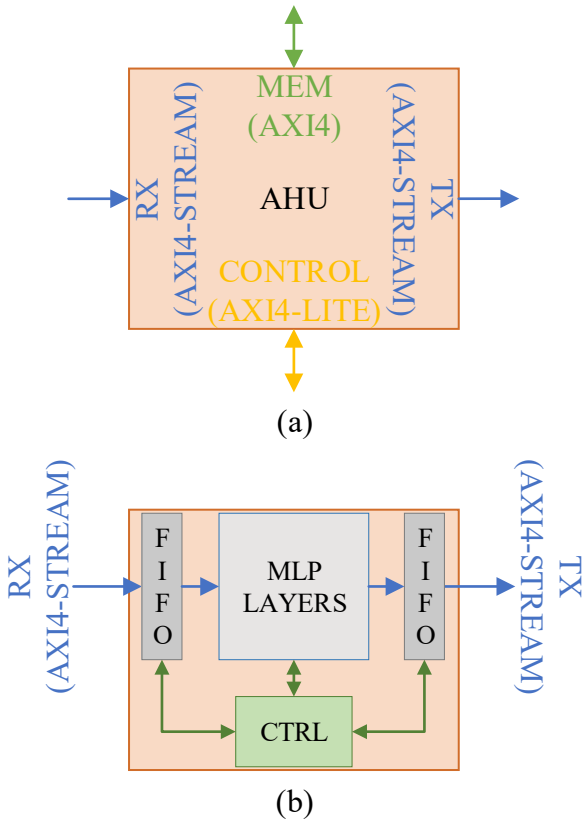

Figure 6.3 *(a) CRUN standard AHU interfaces. (b) AHU for Anomaly Detection neural network RTL kernel referred to as MLP LAYERS.*

bursts from Ethernet. The control logic is designed for controlling the packets arrival and departure from and to the networking portion of the shell.

If a neural network model is too big and cannot fit into only one FPGA on-chip memory, CRUN shell and its networking capabilities enable the distribution of such model. For the distributed CRUN version, MLP LAYERS was divided in two parts: one consisting of LAYER 0 and LAYER 1 and the other comprising LAYER 2 and LAYER 3. Subsequently, each of them was wrapped by AHU interface wrapper to be synthesized into two different FPGAs.

The final architecture of the distributed version is showed in Figure 6.4. Also, MLP LAYERS 1 utilizes the unrolled version of LAYER 2. Only the FPGAs are showed and not their detailed connections.

For a complete overview of the system, Figure 6.1 shows the distributed acceleration with the orange dashed line of route number 2. The two FPGAs in the chain corresponds to FPGA 0 and FPGA 1 of Figure 6.4.

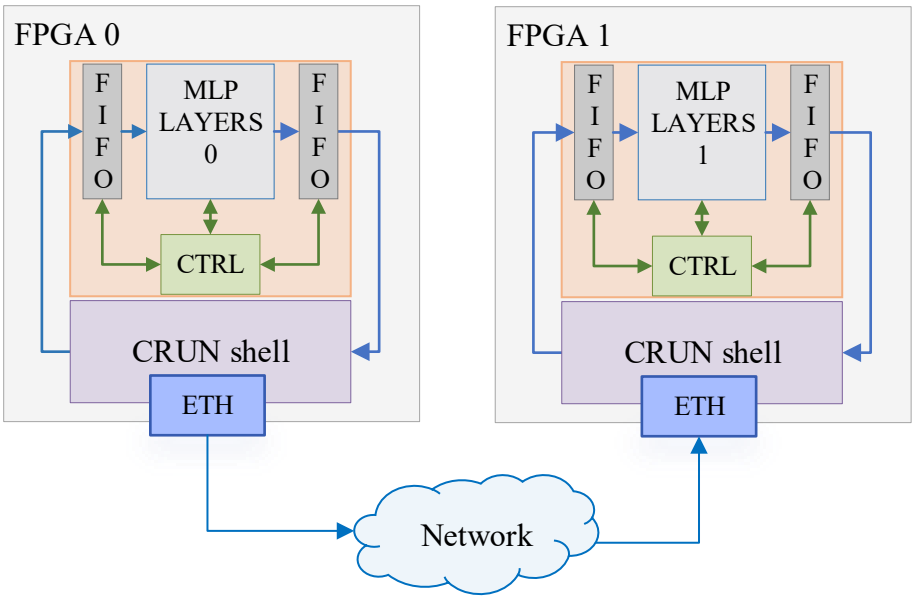

Figure 6.4 *Distributed CRUN implementation.*

6.2 Anomaly Detection MLP

For the subsequent discussion, it is important to address the usage of auto-encoders for anomaly detection. It is sufficient to state that this type of DNNs structures are based upon MLPs, CNNs or RNNs and that it comprises two neural networks, the encoder and decoder. What is different from common types of neural networks is the learning process applied to such auto-encoders [83].

The MLP used here was trained with the method discussed in [4]. Notice that in their work, the authors used the complete auto-encoder, while for the inference phase, as the object of study here, only the encoder portion is needed.

From a high level overview, the MLP used is described in Figure 6.5. This representation shows only the layers and their sizes relation. In this sense, the ReLU nonlinearity between layers is not represented neither the bias summation.

It is important to distinguish two aspects when considering application latency. From the application software perspective, the latency requirement includes the software layer, transport layer and the accelerator processing itself. This constitutes the round-trip latency, the time for sending one or a set of inputs from the software level and receiving the respective output. From the accelerator perspective, the latency reflects the time passed between receiving one input and delivering the correspondent output. In this context, it is crucial to underline that even if

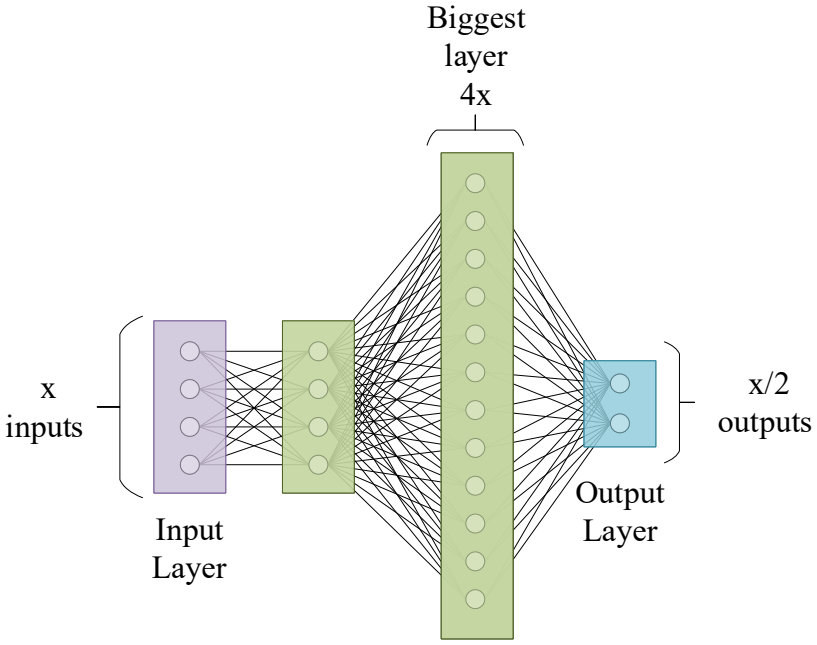

Figure 6.5 *Anomaly detection example application MLP. The layer sizes are depicted as relative to the input size.*

the requirement for the software application seems tight, the accelerator latency requirement is tighter.

For the study demonstrated here the objective was to have an overall application round-trip latency between 20μ and 40μ seconds, which essentially means that the inference accelerator should process one input in less than these values. More details about this will be given in Chapter 7.

6.3 RTL Implementation

For the reasons discussed in Section 3.6, the original anomaly detection model was quantized from 32-bits floating point to 16-bits fixed-point, while the biases were quantized to 32-bits fixed point for accuracy reasons. This means that the inputs, outputs and weights of each layer uses 16-bit representation and only biases utilize 32-bit representation.

Each layer of the MLP showed in Figure 6.5 is formed by Multiply-Accumulation engines, responsible for the multiplication of inputs and weights as well as the summation carried out for each neuron, corresponding to Equation 3.1. These engines process the inputs in parallel but generate outputs sequentially. Each output is

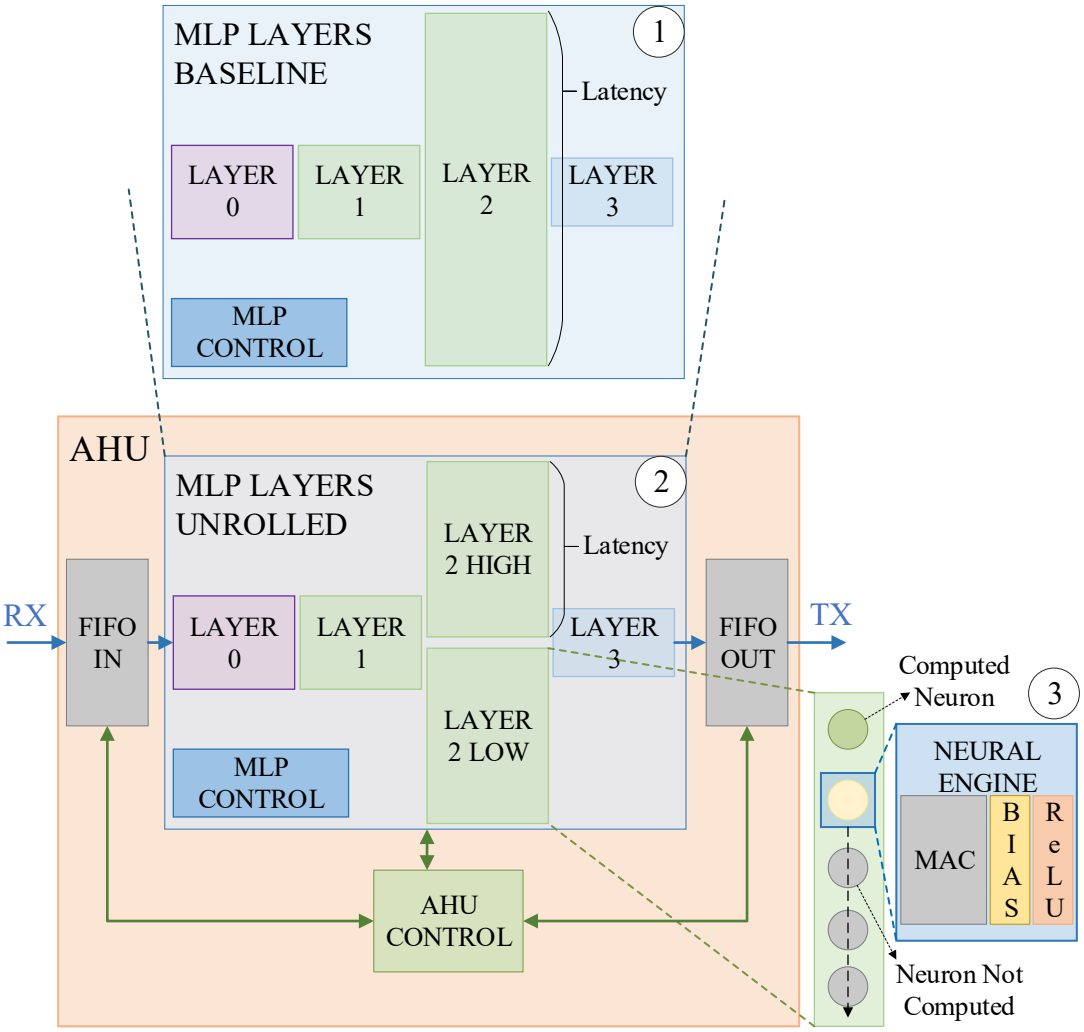

Figure 6.6 *Hierarchical view of Anomaly Layers MLP, in 1 is the original design referred to as baseline and 2 shows the version with the biggest layer unrolled. The computation of neurons of a single layer is displayed in 3. Green represents the neuron already computed, yellow means processing, and gray refers for neurons not yet computed.*

summed with its respective bias and subsequently passed by the ReLU activation function block.

Figure 6.6 shows this behavior in 3. Note that each engine is reused for the computation of each neuron on the network.

From the memory perspective a persistent approach is used here to take advantage of on-chip memory only. As such, distributed ROM memories are inferred from RTL code, whilst weights and biases are fixed at instantiation instead of being loaded during runtime.

The RTL code is configurable on synthesis time and its options are:

- the bit-width of inputs, weights, bias and outputs;
- the number of inputs and outputs;
- usage of the activation function (ReLU);
- the rounding format of the output;
- the weight/bias input files.

A hierarchical overview of the hardware implementation is showed in Figure 6.6. Custom interfaces are used in all levels except when interfacing with the CRUN shell. The operation of each layer simply comprises a start and done signal.

The latency for each layer computation depends directly on the size of the layer. For example, referring to Figure 6.6 in 1, LAYER 2 is four times the size of LAYER 0, hence it takes four times more for computing its outputs.

So far, the discussion was about a single layer, i.e. LAYER 0 in Figure 6.6. One level up there is the addition of the control logic. Since each layer has no back-pressure support because no handshake is implemented, it is responsibility of the control to schedule correctly the inputs for each layer. The output of each layer is the input of the next, i.e. output of LAYER 0 is the input of LAYER 1 and so on.

Two implementations utilize the RTL described here, with some changes regarding their nature. The SDAccel implementation is a co-processor type of acceleration, in which host and FPGA transfer data by Direct Memory Access (DMA). The CRUN implementation is a stream-like type of acceleration, in which the inputs are streamed to FPGA over Ethernet.

The point here is that both implementation utilizes the same RTL kernel with some differences on how the layers operate, their control. Figure 6.7 shows these differences.

For the SDAccel implementation, the control logic starts all layers at the same time. Thus, the design is latency bounded to the biggest layer computation and it is safe to assume that once the biggest layer finishes it is possible to forward the output of each layer. However, this also means that in the beginning only the fourth output should be accepted and the first three discarded. Observe Figure 6.7 (a) for reference.

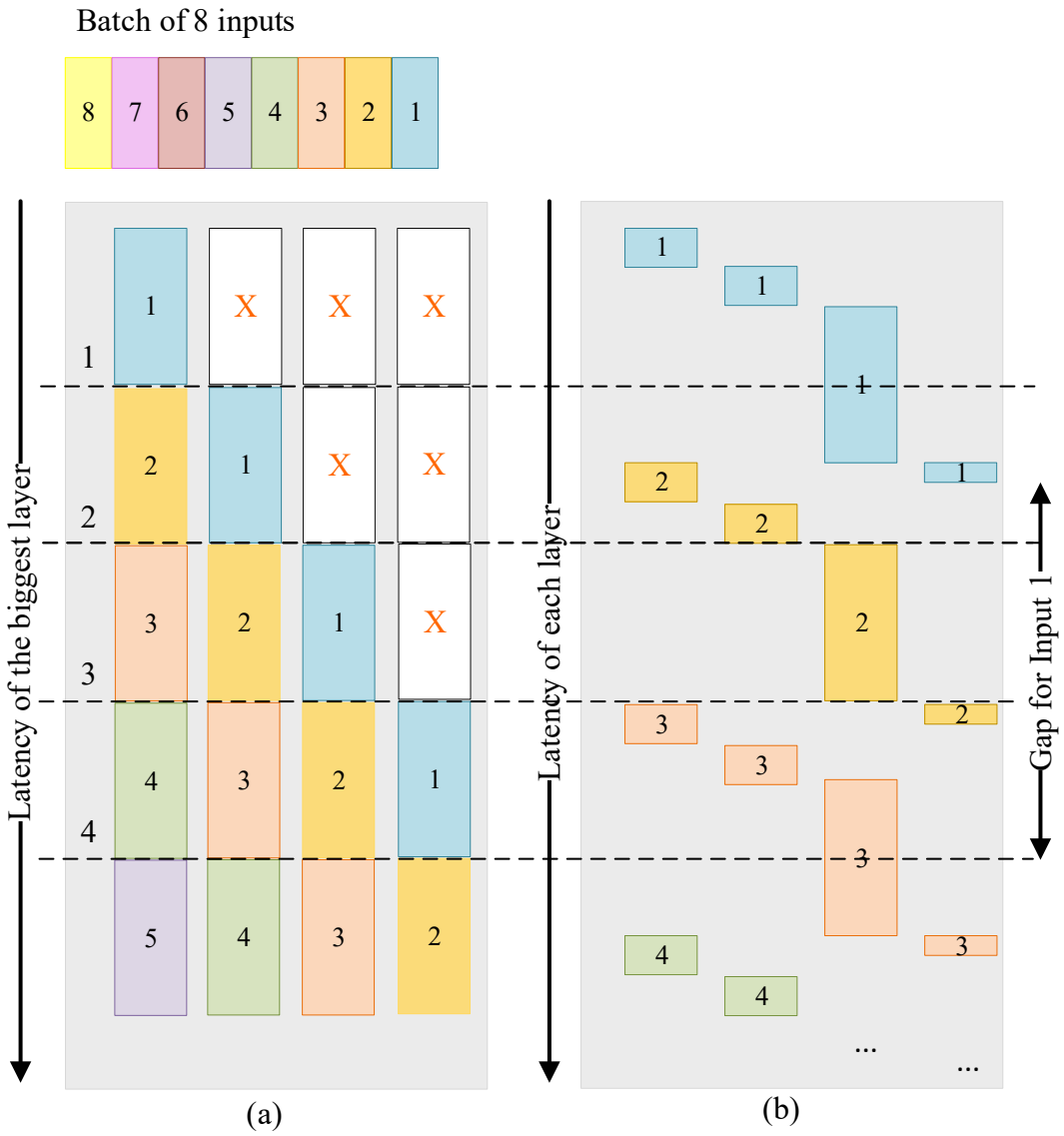

Figure 6.7 *Comparison between different control options. A batch-8 of inputs is showed at the top of the figure. (a) shows the SDAccel operation, since all layers start at the same time, from step 1 to 2 the biggest layer was completed. The latency between each step is the biggest layer. (b) shows the CRUN operation. Each step is correspondent to the latency of each layer. Observe the gap for input 1 between both control options.*

On the other hand, for CRUN implementation, because of the stream-like behavior and no back-pressure support, this is not feasible. Layers have different computation times and the control should be able to start a specific layer at the correct time, when inputs are correctly placed. As a simplistic approach, the control logic starts one layer after the other i.e. first start LAYER 0, once it finishes, start LAYER 1 and so on. When the biggest layer is completed, it is possible to accept more

inputs on LAYER 0. This process is depicted on Figure 6.7 (b). By using the completion of the biggest layer, the approach becomes general for using with other neural networks.

In this context, the biggest layer is the bottleneck of both designs. One possible solution is to unroll the biggest layer at the cost of a bigger area. Figure 6.6 shows this approach in 2. The effect of unrolling the biggest layer was carried out only for CRUN implementation.

At this point, it is possible to summarize the optimizations used in the RTL implementation of anomaly detection neural network, as discussed in Chapter 4.

- Quantization: when quantizing from 32-bit floating point to 16/32-bit fixed point representation.
- Reuse: MAC engines are reused for each neuron output computation.
- Memory hierarchy: persistent approach, only on-chip memory is used for storing weights and biases.
- Layer-parallelism: the biggest layer is unrolled for decreasing latency of computation.

7. RESULTS AND ANALYSIS

In this chapter the results for all mentioned implementations are presented. Analysis and discussion for each solution leveraging pros and cons is subsequently given.

First, a common discussion for all trials is the accuracy drop from the original model to the quantized model. As mentioned previously, CPU and GPU used the original model, while GEMX, SDAccel and CRUN used the quantized version. Note that, GEMX is a high-level implementation, whilst SDAccel and CRUN utilizes a hardware RTL kernel, but they were both validated to the GEMX implementation. More information about validation is given in Section 5.3.

In this regard, it is possible to establish only two different models when referring to accuracy: the original Keras model and the quantized model for RTL implementation. The original model utilized 32-bits floating point representation, while the quantized model 16-bits and 32-bits fixed point representation. Thus, the effect of the quantization on anomaly detection model was reportedly inexpressive for the quantized neural network and the drop in accuracy is less than 0.002%, which is insignificant.

7.1 Performance

Table 7.1 shows the results for five trials of the anomaly detection neural network. Different batch sizes and RTL implementations were used for this comparison. Since the drop in accuracy was insignificant and the quality of the results are all the same, three metrics are used here for evaluating each solution, in order of importance:

- Latency;
- Inferences per second and
- Throughput.

NN Model in Table 7.1 refers to the implementation used. CRUN-B is the Baseline version and it means the RTL anomaly detection neural network discussed in

Table 7.1 *Results for different implementations of anomaly detection neural network.*

Experiment	NN Model	Rep.	Latency/ Batch [μs]	Throug. [Mbps]	Inf. /sec	Batch Size	Freq. [MHz]	FPGA Board
CPU-1	Keras	fp32	798	20.53	1 253	1	NA	NA
CPU-16	Keras	fp32	3 694.6	70.95	4 330	16	NA	NA
GPU-1	Keras	fp32	1 897.43	8.635	527	1	NA	NA
GPU-16	Keras	fp32	1 973.49	132.83	8 107	16	NA	NA
GEMX-32	Python	16-bit	1 500	174.76	21 333	32	60	VCU
SDAccel-16	Baseline	16-bit	602.5	217.55	26 556	16	100	VCU
SDAccel-1	Baseline	16-bit	272.5	30	3 662	1	100	VCU
CRUN-B	Baseline	16-bit	30.97 (+5)	405.5	49 499	1	156.25	KCU
CRUN-U	Unrolled	16-bit	24.40 (+5)	594.48	72 568	1	156.25	KCU
CRUN-D	Unrolled	16-bit	32.55 (+5)	1 232.8	150 488	1	156.25	KCU

Chapter 6. Recall the differences in the operation between SDAccel and CRUN implementation. Unrolled NN model is mentioned as CRUN-U and refers to the version with the biggest layer (LAYER 2) unrolled. CRUN-D is the distributed version of the neural network. CRUN-B and CRUN-U corresponds to the route in Figure 5.1 (a) while CRUN-D is showed in 5.1 (c).

It is important to highlight that the latencies were obtained from the same software level for all solutions. This means that the roundtrip latency provided in Table 7.1 refers to the latency for one input being computed and resulting into one output at the software level. The CRUN latencies presented were calculated from TRex measurements. Figure 5.1 shows the latency measurement points for each case.

The latency reported for the CPU-1 case refers to batch-1. If the batch size is increased to 16 in CPU-16, the latency grows by a factor of 4.6x, but the throughput is only 3.4x bigger, the equivalent to 70 Mbps. The throughput for GPU also increases when comparing batch-1 (GPU-1) and batch-16 (GPU-16), but for this case, the increase is around 16x while the difference on latency is not significant. This means that for large batch sizes the GPU tends to increase the latency more slowly than the CPU does. Likewise, the throughput will increase more rapidly for GPU than for the CPU. For GPU-16 the utilization of the GPU is around 1%, which shows that this small model on GPU with small batch size does not uses the device properly.

The anomaly detection application, particularly is not a good fit for big batches, this is because the latency requirements are extremely tight and only batch-1 can deliver them. Table 7.1 shows only small batches being used, although batch-32 is used for GEMX for achieving the required inferences per second metric. Thus, increasing the batch size for CPU or GPU is not a good alternative. If the concern is purely

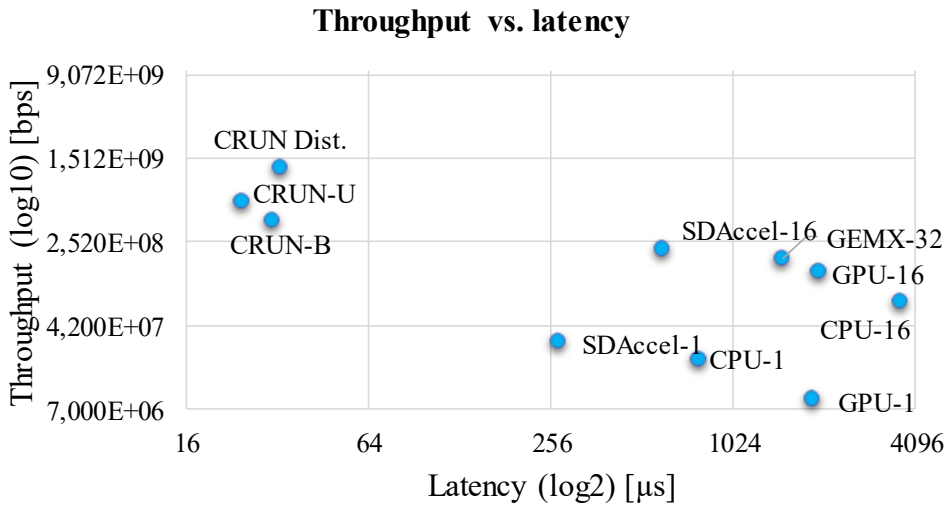

Figure 7.1 *Throughput (bps) vs latency (µs).*

ultra-low latency, neither CPU or GPU can give the best results, even with batch-1. On the other hand, it is clear that CRUN delivers the best throughput and latency. This assertion is true even when comparing two similar implementations, in terms of the RTL kernel used, SDAccel and CRUN. Latency improves 96% when comparing with SDAccel batch-16 (SDAccel-16) and CRUN-U, while the throughput is also improved by 2.7x even when comparing CRUN batch-1 and SDAccel batch-16.

Figure 7.1 presents a graph of throughput vs. latency of the different implementations. CRUN undoubtedly gives the best results and allows ultra-low latency inference for the anomaly detection neural network. It gives more than 30x improvement over CPU-1 latency. Now, observe how the batch size influences on the throughput and the latency of the inference for different solutions.

It is not possible to compete with GPU's throughput when using large batches and an optimized implementation that would use all of its resources. This is because for bigger batch sizes, latency is sacrificed to achieve more bandwidth, which is usually the most pressing issue for demonstrating performance, but not for this thesis. The same can be said for GEMX implementation, where batch size is also used for mitigating the memory accesses by transferring several inputs at a time, instead of just one. Observe that all these solutions are kept on a different latency level from SDAccel and CRUN.

GEMX is a quantized model implementation running on FPGA and shows already a very good performance when compared with CPU and GPU. It uses batch-32 and it gives better latency than CPU and GPU with batch-16. The increase on throughput

is 24% when comparing with GPU-16 while the latency is lowered by the same factor. Also, observe how when comparing inferences per second, the improvement against GPU-16 is by a factor of 2.6, this is an important aspect when comparing different bit-widths, as will be discussed later.

When comparing the results for GEMX, SDAccel and CRUN, one may notice that the FPGA boards used are different. In fact, the FPGA that was used for the first two is bigger and in theory should deliver better results, this is because there is more area for routing and higher frequencies can be achieved. However, notice how the operating frequencies for GEMX and SDAccel are low when compared to CRUN. For GEMX it is 60 MHz, and this cannot be controlled externally on the implementation. The reason behind the 100 MHz of SDAccel is the static shell portion that composes these designs, since the SDAccel shell uses a considerable portion of the FPGA, the RTL kernel needs to be routed around it, which may cause the drop on frequency.

Two trials with SDAccel were carried out with different batch sizes. Observe the difference in latency for batch sizes 16 and 1, while also considering the obvious relationship of batch sizes and throughput. With batch-16 (SDAccel-16) the latency increases only 2.2x while the throughput grows by 7.2x. The reason behind this is exactly the same mechanism behind CPU, GPU and GEMX batch scenarios. From the 272.5μs for batch-1, the correspondent to 245μs is related to memory accesses, which represents almost 90% of the latency. With this mind, it is absolutely clear why to use batches, because of the overhead of moving data back and forth to the FPGA with DDR (Double Data Rate) operation latency cost.

The superior results achieved by CRUN implementation targets exactly the costly DDR operations. Since only on-chip memory is used for storing the neural network model, fair to mention that the exact same approach is taken in SDAccel, the difference is that the input and output data does not need to be transferred via memory. Instead, they are transferred in Ethernet-based packets that leverages DPDK as software acceleration. This alongside the RTL implementation and its optimizations allow the system to achieve the presented latencies.

There are differences between the latencies of the baseline and unrolled versions for CRUN. Those latencies were obtained using TRex and as such they represent the software level round-trip latency.

It is easier to compare baseline and unrolled versions when analyzing the HW-only measurements, which were obtained from clock cycle measurement in simulation and proofed with ILA cores on the design. These latencies are 26.29μs for baseline and 19.72μs for unrolled. For both measurements 4.245μs corresponds to the latency

of the CRUN shell, what gives the latency for only the anomaly detection neural network as 22.04μs for baseline and 15.48μs for unrolled. The throughput values are the theoretical figures and agrees with the measured ones with IXIA.

By unrolling the biggest layer, as showed in Figure 6.6, the latency is dropped by 21%. Observe that the throughput also increases considerably. This means that further unrolling this layer would still improve latency and throughput alike, although the improvement would be smaller and smaller in this process.

There are still two possibilities that could be explored for obtaining even bigger throughput for the anomaly detection neural network computation. First, the operating frequency for the anomaly detection neural network on CRUN could be increased if, for example, the bigger board VCU1525 were used. Although the 156.25 MHz comes from the CRUN shell, there is no limitation for the kernel, only its own computational demands. Second, the operation of the control for the CRUN implementation. It would be possible, by adding handshake signals to each layer, to start the next layer immediately after the previous layer has computed its output, which would double its current throughput to around 1.2 Gbps. This operation change was not pursued because the current values already give a good enough system for this study purposes.

Nevertheless, the HW-only measurements are not a fair comparison with the other solutions, because it does not consider the latency from the SW layer. The measurements with TRex are on Table 7.1. However two different values were obtained utilizing different TRex operation modes. TRex can operate in stateful or stateless mode and the latency measurements were done for both. The difference between these modes represented the addition of 5μs to the values showed in Table 7.1. In this context, it is safe to assume that the latencies for these cases are between the values presented on Table 7.1 and the 5μs addition. The differences in measurements are due to the different flow that TRex utilizes when in stateless mode.

One may argue that this addition is too low compared to the other implementation's latencies. This is true but when considering ultra-low latency scenarios, even the smallest of the values start to make a difference.

Even in the worst case, the results with CRUN are superior. The round-trip latency is between 24μs and 29μs. Important to highlight that the usage of TRex application here is the key for achieving these latencies for data transmissions. Indeed, DPDK, which is used by TRex shows similar results to RDMA in comparison depending on the packet size, as showed in the work by [26]. In that sense, DPDK and RDMA are referred as kernel bypassing technologies and as such they are used for delivering

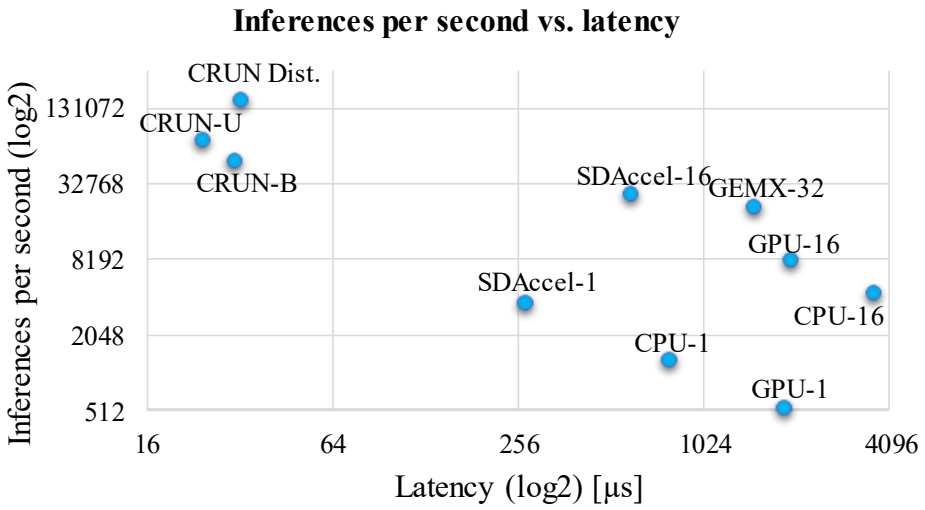

Figure 7.2 *Inference per second vs. latency (µs).*

low-latency communication among the components in a data center [26].

In this context, an application built especially for CRUN would need to use these technologies for optimizing these communications and enabling its usage in a real environment.

Still regarding the throughput presented for different solutions on Table 7.1, one may raise the point that to compare CPU, GPU with the other fixed point implementations is not fair. That is because the former utilizes 32 bits floating-point representation and the throughput is described in bits per second. For that reason, inferences per second metric is used instead of throughput in Figure 7.2.

With this in mind, once again CRUN versions show the best results in latency and throughput. If optimized designs with larger batch sizes are taken into use for GPU implementation, CRUN distributed version may not deliver equivalent performance. However, the most important metric here is not inference per second, but latency, and in this figure any CRUN implementation is undoubtedly the best.

One down side of CRUN distributed version is the utilization of more area by using two FPGAs. From the bright side, it gives a 98% drop on latency, while giving 18.5x increase for inference per second than GPU batch-16.

7.2 Resource Utilization

Also mentioned in Chapter 6, the decrease on latency comes at a cost, which is area. To underline this trade-off, Table 7.2 shows the resource usage for both versions of the RTL implementation and for CRUN shell only design.

Table 7.2 *Resource utilization for anomaly detection NN versions.*

Resource	**Utilization %**		
	Baseline	**Unrolled**	**Shell Only**
LUT	37,22	40,67	15,41
LUTRAM	5,54	5,79	5,53
FF	48,12	51,13	13,56
BRAM	47,96	57,01	17,25
DSP	64,93	74,20	0

First, one can acknowledge the size of the CRUN shell and how it restrains the AHU design. Especially important to notice is that almost 18% of Block RAMs (BRAMs) are already in use by the shell, which is a primary concern regarding neural networks model sizes. Another consideration is about the DSP usage, CRUN shell does not utilize any DSP resource which is of prime importance when implementing the needed multiplications in a neural network computation.

Furthermore, the unrolled version had an increase of around 10% for BRAM and DSP. One can argue that this is a small price to pay for obtaining 79% of the system latency. But in that case, depending on the neural network, this will not be possible with only one FPGA, which is exactly the argument for distributing it across two or several FPGAs.

In this case, the anomaly detection neural network was distributed across two FPGAs. Some considerations are important in this scenario. The CRUN shell latency is doubled and impacts more heavily the overall latency, contributing to roughly half the latency of the anomaly detection computation itself. Also, the latencies of the switch (already included on all the latencies for CRUN) will also contribute more because of the added routing path.

In summary for a two FPGA design, the shell and the switch latency from a one FPGA design would be doubled. From the bright side, with more area the biggest layer could be unrolled once more as already discussed. In summary, even for low latency use cases, this mode could still be applicable if more throughput is desired.

7.3 Design Complexity

The complexity of the implementation can also be inferred from Table 7.1. The presentation of solutions is in increasing order of complexity. Indeed, once one has the Keras model, it is very straightforward to run the inference phase with it. That is the reason why CPU and GPU are at the top of the table, since it is basically a matter of calling the predict method for the model. When it comes to running the predict method on CPU or GPU, it can be basically done with a few lines of code in a Python script by directly calling TensorFlow functions.

GEMX implementation is one step further on the difficulty scale. Although this was carried out by using a Python API for GEMX, it still imposes some challenges. The quantization is basically one of them. However, since no RTL is directly needed, and as such, a Python script can handle basically the whole process, it still constitutes an easier implementation.

SDAccel represents a new level of complexity. As mentioned there is the possibility of using standard programming languages or even HLS for describing the computational kernel. However, in this case, for the optimum solution, RTL kernel was implemented from scratch. This can be easily justified by the extent in which optimizations can be done at this level of abstraction. Nevertheless, SDAccel leverages a standard type of acceleration, in which data is written to the device memory, the computation is carried out and once it is completed the host reads the result from the DRAM of the device. This also means that the framework provides a good support for handling this process, offering wizards for creating the necessary interfaces for communication with SDAccel shell.

At the other end on the complexity spectrum is CRUN implementation. CRUN uses basically the same underlying hardware as SDAccel and as such already imposes a great challenge for RTL kernel development. The difference here is that no wizard can be used for generating the necessary interfaces for the CRUN shell, which complicates the process. Another important point is the difference regarding the acceleration mode, since CRUN utilizes Ethernet packets, this should be handled from SW level, and as already mentioned TRex was used for this purpose with custom payload data.

Another thought when distributing the design is the synthesis and implementation time that is reduced proportionally to size. In this case since the design was separated into two, the synthesis design for each is less than it was for the whole design, which is also a point to consider. However, this consideration may have a minor impact if this procedure is seldom executed.

7.4 Limitations

The results from CPU and GPU does not contain any optimization. This means that they were used plainly from Keras framework and TensorFlow backend. In fact, it is possible that if these solutions were implemented with more effort the results could be improved. However, notice that a tailored implementation targeting better values increases the complexity.

When it comes to the limitations, this is an off-line experiment. Data for the anomaly detection NN algorithm was already preprocessed and the latencies for doing so are not included in these results. This is done because of the testing mode employed, with TRex. Since the measurements are carried out for all implementations from the same level, the preprocessing is excluded from all trials, making the results comparable.

Another limitation constitutes the flexibility of this design. The anomaly detection neural network was especially designed for this purpose. This means that if another neural network is required, several modifications would need to be made, although the underlying RTL has a level of generalization, no convolutional layer was developed for example. Yet, this lack of flexibility is exactly one of the reasons for the excellent results when referring to latency, because of the optimizations implemented.

Also note that the system implemented and used with CRUN cannot be applied to every neural network. In the case of the anomaly detection NN since an MLP was used this stream-like approach is a perfect fit. However, if complex layers or short-cut connections are needed, this approach cannot be used.

From the measurements with TRex for CRUN, the figures on Table 7.1 are average values. However, the maximum latency observed was 290 μs. This means, that in order to have a reliable system this should be taken into account.

Finally, the latencies for CRUN cases were calculated from individual measurements, CRUN shell only, AHU and TRex. This can potentially be translated as an error source, but it is not expected to invalidate these results. One important consideration in this aspect is that the HW measurements are completely reliable.

8. CONCLUSIONS

In this thesis work, a comparison between five implementations of an anomaly detection neural network inference was studied.

It is clear that the best performing solution in terms of latency and throughput is provided by CRUN unrolled version. This means that a hardware neural network implementation leveraging several optimizations allied with CRUN framework offers the possibility of running inference in the data center environment with ultra-low latency.

The requirements for this solution were to obtain latency between 20μs to 40μs for inference time and 20 000 inferences per second. These goals were categorically fulfilled with all CRUN implementations, even with the worst performing solution, that was the baseline version.

The improvement in performance is also observed when comparing similar implementations, in terms of the RTL kernel used, as is the case of SDAccel, the second-best solution after CRUN. Latency improves 96% when comparing with SDAccel batch-16 and the throughput is also improved by 2.7x even when looking at CRUN batch-1 and SDAccel batch-16.

This is especially important in the context of mobile networks and the edge cloud. The impact of this study has a prime importance within 5G scope. This is because, depending on the application, the deployment of deep learning solutions requires a low latency format that may not be achieved when using expensive memory to memory communications, but is facilitated when utilizing stream-like style. The innovative approach of utilizing Ethernet based packet communications for deep learning acceleration, although not new even in cloud environment, is a pioneer on the mobile networks cloud context.

As a general guideline, CRUN should be used when ultra-low latencies are required, in which batch-1 cases is the only solution for fulfilling this requirement. However, its usability is tied to the type of application and neural network used, which should be heavily considered. Nothing else can be inferred for the usage of CRUN because

more experiments are needed to draw deeper conclusions.

For future work, optimizations for achieving even lower latencies will be done. This is needed because the better the latency of the neural network acceleration the more time is spared for communication and processing tasks, what would impact the overall performance of the system.

BIBLIOGRAPHY

[1] "IEEE Standard for Ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, March 2016.

[2] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A Survey," 05 2018.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[4] C. Aytekin, X. Ni, F. Cricri, and E. Aksu, "Clustering and Unsupervised Anomaly Detection with L2 Normalized Deep Auto-Encoder Representations," *CoRR*, vol. abs/1802.00187, 2018.

[5] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling Up Machine Learning: Introduction*. Cambridge University Press, 2011.

[6] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[7] M. Blott, T. B. Preusser, N. C. Fraser, G. Gambardella, K. O'Brien, and Y. Umuroglu, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," 2018.

[8] M. Blott, T. B. Preusser, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, and M. Leeser, "Scaling neural network performance through customized hardware architectures on reconfigurable logic," *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 419–422, 2017.

[9] G. Brown and H. Reading, *Cloud RAN & the Next-Generation Mobile Network Architecture, White Paper*.

[10] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016.

[11] C. Chang, N. Nikaein, R. Knopp, T. Spyropoulos, and S. S. Kumar, "FlexCRAN: A flexible functional split framework over ethernet fronthaul in Cloud-RAN," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–7.

[12] A. Checko, A. P. Avramova, M. S. Berger, and H. L. Christiansen, "Evaluating C-RAN fronthaul functional splits in terms of network level energy and cost savings," *Journal of Communications and Networks*, vol. 18, no. 2, pp. 162–172, April 2016.

[13] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann, "Cloud ran for mobile networks a technology overview," *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 405–426, Firstquarter 2015.

[14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 609–622.

[15] K. Cheung. How nokia is utilizing machine learning in 5g networks. [Online]. Available: https://algorithmxlab.com/blog/2018/06/27/how-nokia-is-utilizing-machine-learning-in-5g-networks-2/

[16] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[17] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, M. Ghandi, D. Lo, S. Reinhardt, S. Alkalay, H. Angepat, D. Chiou, A. Forin, D. Burger, L. Woods, G. Weisz, M. Haselman, and D. Zhang, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave." IEEE, March 2018. [Online]. Available: https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/

[18] Cisco, "Global mobile data traffic forecast update." [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html

[19] Cisco. Trex. [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_book.pdf

[20] F. Conti, P. D. Schiavone, and L. Benini, "XNOR Neural Engine: a Hardware Accelerator IP for 21.6 fJ/op Binary Neural Network Inference," *CoRR*, vol. abs/1807.03010, 2018.

[21] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, "Advancing software-defined networks: A survey," *IEEE Access*, vol. 5, pp. 25 487–25 526, 2017.

[22] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO*, 2017.

[23] Y. Ding, S. Hu, M. T. Niemier, J. Cong, Y. H. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," 2018.

[24] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 1–14.

[25] R. Garg and L. Hendren, "A portable and high-performance general matrix-multiply (gemm) library for gpus and single-chip cpu/gpu systems," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2014, pp. 672–680.

[26] D. Gehberger, D. Balla, M. Maliosz, and C. Simon, "Performance Evaluation of Low Latency Communication Alternatives in a Containerized Cloud Environment," *3D Digital Imaging and Modeling, International Conference on*, pp. 9–16.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[28] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 152–159, 2017.

[29] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 35–47, 2018.

[30] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA based neural network accelerator," *CoRR*, vol. abs/1712.08934, 2017. [Online]. Available: http://arxiv.org/abs/1712.08934

[31] A. Hadnagy, B. Fehér, and T. Kovácsházy, "Efficient implementation of convolutional neural networks on FPGA," in *2018 19th International Carpathian Control Conference (ICCC)*, May 2018, pp. 359–364.

[32] D. Harutyunyan and R. Riggio, "Flexible functional split in 5G networks," in *2017 13th International Conference on Network and Service Management (CNSM)*, Nov 2017, pp. 1–9.

[33] M. Huang, X. Wang, K. Li, and S. K. Das, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018.

[34] INTEL. Intel Xeon Gold 6130 Processor. [Online]. Available: https://ark.intel.com/products/120492/Intel-Xeon-Gold-6130-Processor-22M-Cache-2-10-GHz-

[35] ITU-T, *Transport network support of IMT-2020/5G*.

[36] C. Jiang, H. Zhang, Y. Ren, Z. Han, K. Chen, and L. Hanzo, "Machine learning paradigms for next-generation wireless networks," *IEEE Wireless Communications*, vol. 24, no. 2, pp. 98–105, April 2017.

[37] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. K. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[38] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1462–1467, 2017.

[39] J. H. Kim, B. Grady, R. Lian, J. Brothers, and J. H. Anderson, "Fpga-based cnn inference accelerator synthesized from multi-threaded c software," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, Sept 2017, pp. 268–273.

[40] E. J. Kitindi, S. Fu, Y. Jia, A. Kabir, and Y. Wang, "Wireless network virtualization with sdn and c-ran for 5g networks: Requirements, opportunities, and challenges," *IEEE Access*, vol. 5, pp. 19 099–19 115, 2017.

[41] D. Koslopp, "CRUN: Distributed Processing in FPGA Accelerated Cloud," Master of Science Thesis, Tampere University of Technology, 2018.

[42] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[44] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," *CoRR*, vol. abs/1602.04283, 2016. [Online]. Available: http://arxiv.org/abs/1602.04283

[45] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-x : An accelerator architecture for exploiting unstructured sparsity in deep neural networks," 2018.

[46] Z. Li, Y. Wang, T. Zhi, and T. Chen, "A survey of neural network accelerators," *Front. Comput. Sci.*, vol. 11, no. 5, pp. 746–761, Oct. 2017. [Online]. Available: https://doi.org/10.1007/s11704-016-6159-1

[47] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405, 2016.

[48] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, vol. 00, April 2017, pp. 101–108. [Online]. Available: doi.ieeecomputersociety.org/10.1109/FCCM.2017.64

[49] Y. Ma, Y. Cao, S. B. K. Vrudhula, and J. sun Seo, "An automatic rtl compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2017.

[50] A. Maeder, M. Lalam, A. D. Domenico, E. Pateromichelakis, D. WÃ$\frac{1}{4}$bben, J. Bartelt, R. Fritzsche, and P. Rost, "Towards a flexible functional split for cloud-RAN networks," in *2014 European Conference on Networks and Communications (EuCNC)*, June 2014, pp. 1–5.

[51] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," Gaithersburg, MD, United States, Tech. Rep., 2011.

[52] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "Neuraghe: Exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs," *CoRR*, vol. abs/1712.00994, 2017.

[53] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, Firstquarter 2016.

[54] S. N. Motade and A. V. Kulkarni, "Channel estimation and data detection using machine learning for mimo 5g communication systems in fading channel," *Technologies*, vol. 6, no. 3, 2018. [Online]. Available: http://www.mdpi.com/2227-7080/6/3/72

[55] M. A. Nielsen, "Neural networks and deep learning," 2018. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[56] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in nfv environments," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, March 2015, pp. 1–5.

[57] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "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*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021740

[58] NVIDIA. NVDLA open source project. [Online]. Available: http://nvdla.org/

[59] NVIDIA, *NVIDIA Tesla V100 GPU Accelerator*.

[60] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, Dec 2017.

[61] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40, 2017.

[62] M. Patel, Y. Hu, P. Hede, J. Joubert, C. Thornton, B. Naughton, J. R. Ramos, C. Chan, V. Young, S. J. Tan, D. Lynch, N. Sprecher, T. Musiol, C. Manzanares, U. Rauschenbach, S. Abeta, L. Chen, K. Shimizu, A. Neal, P. Cosimini, A. Pollard, and G. Klas, *Mobile-Edge Computing - Introductory Technical White Paper.*

[63] L. Pierucci and D. Micheli, "A Neural Network for Quality of Experience Estimation in Mobile Communications," *IEEE MultiMedia*, vol. 23, no. 4, pp. 42–49, Oct 2016.

[64] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[65] A. Reznik, L. M. C. Murillo, Y. Fang, W. Featherstone, M. Filippou, F. Fontes, F. Giust, Q. Huang, A. Li, C. Turyagyenda, C. Wehner, and Z. Zheng, *Cloud RAN and MEC: A Perfect Pairing White Paper*, ETSI.

[66] W. River, *vRAN: The Next Step in Network Transformation, White Paper.*

[67] R. Rojas, *Neural Networks: A Systematic Introduction.* Berlin, Heidelberg: Springer-Verlag, 1996.

[68] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[69] I. H. C. . Symposium, Ed., *Accelerating Persistent Neural Networks at Datacenter Scale.* [Online]. Available: https://www.hotchips.org/wp-content/uploads/

hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.60-NeuralNet1-Pub/HC29.22,622-Brainwave-Datacenter-Chung-Microsoft-2017_08_11_2017.pdf

[70] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[71] TensorFlow. Tensorflow. [Online]. Available: https://www.tensorflow.org/

[72] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021744

[73] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[74] Xilinx. General matrix operation library. [Online]. Available: https://github.com/Xilinx/gemx

[75] Xilinx. Integrated logic analyzer (ila). [Online]. Available: https://www.xilinx.com/products/intellectual-property/ila.html

[76] Xilinx, *KCU1500 Board User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/kcu1500/ug1260-kcu1500-data-center.pdf

[77] Xilinx, *SDAccel Environment User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1023-sdaccel-user-guide.pdf

[78] Xilinx, *VCU1525 Reconfigurable Acceleration Platform*. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf

[79] Xilinx, *Versal Architecture and Product Data Sheet: Overview*.

[80] Xilinx. Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html#documentation

[81] X. Xu, Y. Ding, S. Xiaobo Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," vol. 1, 04 2018.

[82] H. Ye, G. Y. Li, and B. Juang, "Power of deep learning for channel estimation and signal detection in ofdm systems," *IEEE Wireless Communications Letters*, vol. 7, no. 1, pp. 114–117, Feb 2018.

[83] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *CoRR*, vol. abs/1803.04311, 2018. [Online]. Available: http://arxiv.org/abs/1803.04311

[84] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. F. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[85] R. Zhao, H.-C. Ng, W. Luk, and X. Niu, "Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA," 2018.

[86] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel, "A hybrid asic and fpga architecture," in *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, Nov 2002, pp. 187–194.