



Design of multi-FPGAs platform in the cloud for neural network applications

Wu Chen

► To cite this version:

Wu Chen. Design of multi-FPGAs platform in the cloud for neural network applications. Micro and nanotechnologies/Microelectronics. Université de Lyon, 2021. English. NNT : 2021LYSES030 . tel-03720535

HAL Id: tel-03720535

<https://theses.hal.science/tel-03720535>

Submitted on 12 Jul 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



N° d'ordre NNT : 2021LYSES030

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée conjointement
Université Jean Monnet

École Doctorale 488
Sciences Ingénierie Santé

Spécialité
Micro-électronique

Soutenance publique le 29 Octobre 2021, par :
Chen WU

Design of multi-FPGAs platform in the cloud for neural network applications

Devant le jury composé de :

M. Stéphane Mancini	MCF-HDR	Grenoble INP- UGA	Rapporteur
M. Stefan Duffner	MCF-HDR	Laboratoire LIRIS-INSa Lyon	Rapporteur
M. Dominique Houzet	PR	Lab GIPSA-UGA	Examineur
M. Jean-Paul Jamont	PR	LCIS-UGA	Examineur
Mme. Yang Fan	PR	ImVIA-Université de Bourgogne	Examineur
M. Florent Carlier	MCF- HC	CREN-Le Mans Université	Invité
Mme. Virginie Fresse	MCF-HDR	LaHC-UJM	Directrice de thèse
M. Hubert Konik	MCF	LaHC-UJM	Co-directeur de thèse

Contents

CHAPTER 1 INTRODUCTION	1
1.1 GENERAL CONTEXT	2
1.2 OBJECTIVE	3
1.3 CONTRIBUTIONS	4
1.4 ORGANIZATION OF MANUSCRIPT	5
 CHAPTER 2 CNNs ON FPGA	 7
2.1 CONVOLUTIONAL NEURAL NETWORK	8
2.1.1 Network layers	9
2.1.2 Inference and training	11
2.2 FIELD PROGRAMMABLE DEVICES	14
2.3 DESIGN FLOW OF APPLICATION OF FPGA	14
2.3.1 System specification	14
2.3.2 IP design	15
2.3.3 IP integration	15
2.3.4 Synthesis	17
2.3.5 Placement & routing	17
2.3.6 Programming FPGA	18
2.4 CHALLENGES OF CNNs ON FPGAs	18
 CHAPTER 3 ACCELERATING CNNs FROM LOCAL TO VIRTUALIZED FPGA IN THE CLOUD: A SURVEY OF TRENDS	 21
3.1 INTRODUCTION	22
3.2 BACKGROUND	26
3.2.1 FPGA cloud	26
3.2.2 FPGA virtualization	27
3.3 CNN IMPLEMENTATION TECHNIQUES	30

3.3.1	Hardware architecture design	30
3.3.2	Network compression	31
3.3.3	Optimization strategy	32
3.4	ACCELERATING CNNs FROM LOCAL TO VIRTUALIZED FPGAs IN THE CLOUD	33
3.4.1	CNNs on local FPGA	33
3.4.2	CNNs on local virtualized FPGA	36
3.4.3	CNNs on virtualized FPGA in the cloud	38
3.4.4	CNN deployment in commercial cloud	40
3.5	TRENDS OF CNN ACCELERATORS	40
3.6	DISCUSSION	45
3.6.1	Unresolved challenges	45
3.6.2	Industrial solution	46
3.6.3	Roadblocks of FPGA Cloud	47
3.7	CONCLUSION	47
CHAPTER 4 ACCELERATING CNNs ON FPGA PLATFORM		49
4.1	PRINCIPLE OF THE PLATFORM	51
4.2	STREAMING	53
4.2.1	IP Design	53
4.2.2	Mathematical model of resource utilization	59
4.2.3	Latency results	66
4.3	SINGLE ENGINE COMPUTATION	68
4.3.1	IP design	68
4.3.2	Generic CNN parameters	71
4.3.3	Optimization	71
4.4	QUANTIZATION	73
4.4.1	Our quantization tool	75
4.4.2	Experimental results	79
4.5	CONCLUSION	82
CHAPTER 5 PROPOSAL OF A FPGA-BASED CLOUD PLATFORM		83
5.1	OVERVIEW OF CNN-BASED APPLICATION LIFECYCLE	85
5.1.1	Requirement analysis	86
5.1.2	Data Preparation	87
5.1.3	CNN design	88

5.1.4 CNN deployment	88
5.1.5 Evaluation	89
5.1.6 Production	89
5.1.7 Maintenance	90
5.1.8 Withdrawal	91
5.2 REVIEWING CNN DEPLOYMENT TOOLS IN FPGA CLOUDS.	91
5.2.1 Identified tools	91
5.2.2 Analysis of the tools according to the CNN application lifecycle . . .	92
5.2.3 Challenges	94
5.3 A FRAMEWORK FOR FPGA-BASED CNN DEPLOYMENT PLATFORM	97
5.3.1 Contract manager	99
5.3.2 Architecture manager	99
5.3.3 Monitoring manager	100
5.3.4 Resource manager	101
5.3.5 Storage manager	102
5.4 STUDY CASE	103
5.5 CONCLUSION	107
CHAPTER 6 CONCLUSION AND PERSPECTIVES	109
6.1 CONCLUSION	110
6.2 PERSPECTIVES	110
LISTE DES CONTRIBUTIONS	113
LISTE DES CONTRIBUTIONS	113
LIST OF FIGURES	114
LIST OF TABLES	116
BIBLIOGRAPHY	117

INTRODUCTION

Contents

1.1	GENERAL CONTEXT	2
1.2	OBJECTIVE	3
1.3	CONTRIBUTIONS	4
1.4	ORGANIZATION OF MANUSCRIPT	5

1.1 GENERAL CONTEXT

This thesis is realized in the Hubert Curien Laboratory, supported by China Scholarship Council (Grant number, 201708070009).

In the field of deep learning, neural networks such as convolution neural networks (CNNs) and recurrent neural networks (RNNs) are widely used in computer vision, speech recognition, natural language processing, and other fields [25].

Since 2010, ImageNet ILSVRC Challenge[3] has been held annually to evaluate neural network algorithms for object detection and classification. Each participating group used a large-scale image data set to test the algorithm's performance in processing image classification, target positioning, target detection, and other applications. In 2012, the CNN algorithm achieved a historic breakthrough, where AlexNet[80] achieved a top-5 error of 15.3%, more than 10.8 % lower than that of others participating. CNN's take this breakthrough as an opportunity to promote the development of the current deep learning boom.

The rapid development of neural network algorithms has led to neural network architectures with more calculations and deeper structures. In addition, with the deepening of neural network, the execution of network on the CPU/GPU faces the problem of energy efficiency. Due to Moore's Law, the transistor scale of chips has reached its limit, resulting in slower growth in processor performance[118]. At the same time, because the CPU/GPU consumes too much energy (85W and 200W, respectively) [39], it is not feasible to increase the number of processors to improve execution capabilities. Therefore, the current problem is to reduce energy consumption while maintaining the high efficiency of network execution.

In this case, Field Programmable Gate Array (FPGA) has become another option for implementing neural network algorithms because of its lower energy consumption compared to CPU/GPU [4]. In addition, FPGA has the advantage of being reconfigurable and can be flexibly configured according to the characteristics of the neural network algorithm to meet diverse needs. Finally, FPGA supports multiple granular parallelisms, which means multi-core or many-core can be used to obtain coarse-grained parallelism for neural network acceleration. Therefore, it is possible to select the flexible data bit width of the arithmetic unit for neural network inference without wasting resources.

Meanwhile, cloud computing is considered a technology that can provide end-users

with high-speed, cost-effective, and scalable computing [104]. In the cloud computing scenario, users upload data to a cloud server and send task requests via the Internet. The cloud server processes the user data and returns the result. This calculation method improves the utilization of computing resources and reduces the maintenance cost of software and hardware resources for individual users.

Due to the advantages of cloud computing, many companies are working hard to deploy CPUs/GPUs in the cloud to improve computing power [66, 48, 13]. In 2002, Amazon introduced the Amazon Web Service [16], which allows individual users to access the computation resources in the cloud. In 2010, Microsoft launched the Azure [21] project to provide the facilities of the computing resources and services, which the users can dynamically use according to the application demands. However, deploying FPGAs in the cloud has less success than GPUs because the use of FPGAs in the cloud requires the user's hardware expertise and low-level cognition, which cannot be mastered in a short time. Therefore, only a few companies have put effort into deploying FPGAs in the cloud, such as Amazon AWS [1], HUAWEI cloud [2].

Porting FPGAs as the calculation resource in cloud computing benefits the following advantages: 1) calculation acceleration, the performance, and power benefits are achieved by designing custom calculations of data paths tailored to the application; 2) cloud security, allowing data to be stored and manipulated remotely in an encrypted form, effectively preventing the server from accessing the processed information; 3) lower energy consumption, using FPGAs as a calculation node and realize accelerators based on the FPGA in the cloud.

Therefore, this thesis aims to provide cloud users with an FPGA-based infrastructure to execute their CNN applications in various use cases (for example, CNN model exploration, network parameterization, etc.), not just perform CNN inference.

1.2 OBJECTIVE

The objective of the thesis is to provide a cloud computing infrastructure in which there are several FPGAs of different families and types. These FPGAs communicate with each other and are made available to the machine learning engineer to execute CNNs.

This Cloud FPGA is dedicated to machine learning engineers when they wish to "execute" CNNs, that is to say, to perform training or inference on different CNN models

with image databases and without hardware expertise. The Cloud platform can generate different CNNs architectures and deploy these architectures in the appropriate FPGAs. Additionally, machine learning engineers can exploit these FPGAs by exploring the CNN model and available FPGAs in the cloud.

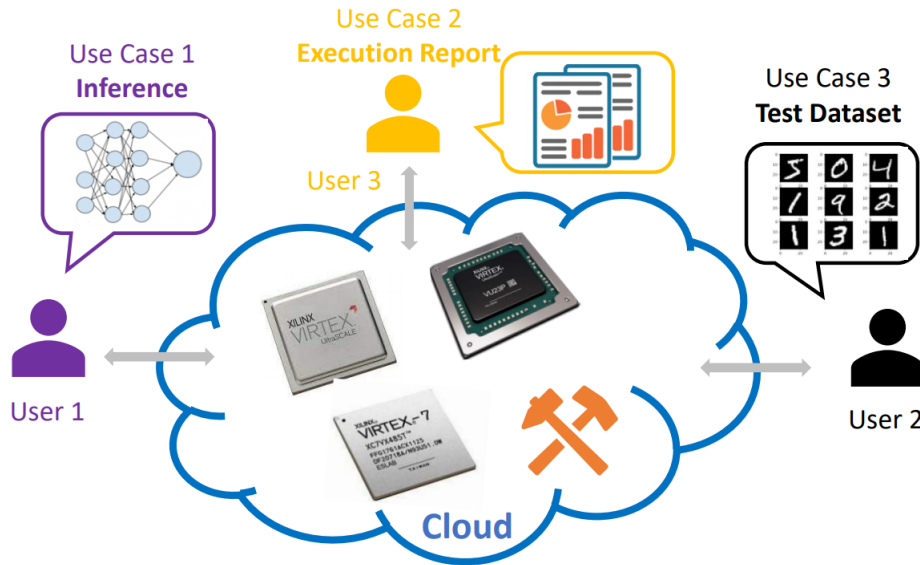


Figure 1.1 - Cloud computing-type platform for CNNs inference on the FPGA.

1.3 CONTRIBUTIONS

The contribution of this thesis is to propose a cloud computing type platform for machine learning engineers to perform general CNN inference on FPGA. Furthermore, the proposed platform can analyze the user needs of different use cases, deploy the CNN hardware architecture on the appropriate FPGA, and implement optimization techniques when necessary.

Two types of generic CNN IPs are defined and developed for generating CNN hardware architectures on the FPGA. These IPs are adapted to several CNNs algorithms by configuring the different CNN parameters. The CNN IPs are tested and integrated by the test vehicle on the FPGA SoC. A communication prototype between Programme logic (PL) and Process system (PS) of the FPGA is achieved in the design.

The platform involves several mathematical models, which estimate the resource usage of the two IPs developed. This estimation can help allocate FPGA resources well in the cloud. A quantization tool is developed to compress the network size on the FPGA. Then, a life cycle of the cloud platform is designed to show the process of executing CNN

inference. This life cycle includes different stages to clarify the execution process of multi-user CNN development.

1.4 ORGANIZATION OF MANUSCRIPT

This manuscript first starts by presenting the general context and objectives of the thesis.

Chapter 2 provides a comprehensive introduction to CNN and FPGA and implementation methods and related challenges.

Chapter 3 conducts in-depth research of the technologies and the evolution of CNN accelerators, from a single CNN in the local to multiple CNNs IN the cloud, enhancing the current understanding of the evolution of FPGA-based CNN accelerators.

Chapter 4 presents the realization of different CNN hardware architectures and the completion of quantification tools and optimization methods.

Chapter 5 proposes a novel platform with a lifecycle for multiple users to execute CNN on FPGA in the cloud.

Chapter 6 concludes this manuscript and presents the future directions of the work.

CNNs on FPGA

Contents

2.1	CONVOLUTIONAL NEURAL NETWORK	8
2.1.1	Network layers	9
2.1.2	Inference and training	11
2.2	FIELD PROGRAMMABLE DEVICES	14
2.3	DESIGN FLOW OF APPLICATION OF FPGA	14
2.3.1	System specification	14
2.3.2	IP design	15
2.3.3	IP integration	15
2.3.4	Synthesis	17
2.3.5	Placement & routing	17
2.3.6	Programming FPGA	18
2.4	CHALLENGES OF CNNs ON FPGAs	18

This thesis proposes a cloud computing type platform for machine learning engineers to perform general CNN usages on FPGA. Therefore, the first step of this platform is to implement CNNs on the FPGA.

The development of CNNs, which involves hundreds of millions of learnable parameters, puts a higher demand on hardware performance. Therefore, how to implement high-performance CNNs on the FPGA-based device has been a new challenge. Besides, under the condition of ensuring performance, how to reduce the calculation and data workload is also an important aspect.

This chapter is organized as the following:

- Section 2.1 gives an introduction to CNN, including the explanation of CNN operation, as well as the training and inference phase in CNN;
- Section 2.2 presents the FPGA devices with the internal architecture and resources;
- Section 2.3 summaries the FPGA design flow, which comprises of several different steps or phases to finally execute an application;
- Section 2.4 lists several significant challenges for CNN's implementation on the FPGAs.

2.1 CONVOLUTIONAL NEURAL NETWORK

Convolutional Neural Network (CNN) is a famous neural network structure in the field of deep learning. Nowadays, many visual image applications apply the CNNs because multi-layer convolution has an impact on the feature extraction of three-dimensional images (RGB) [68, 130, 105].

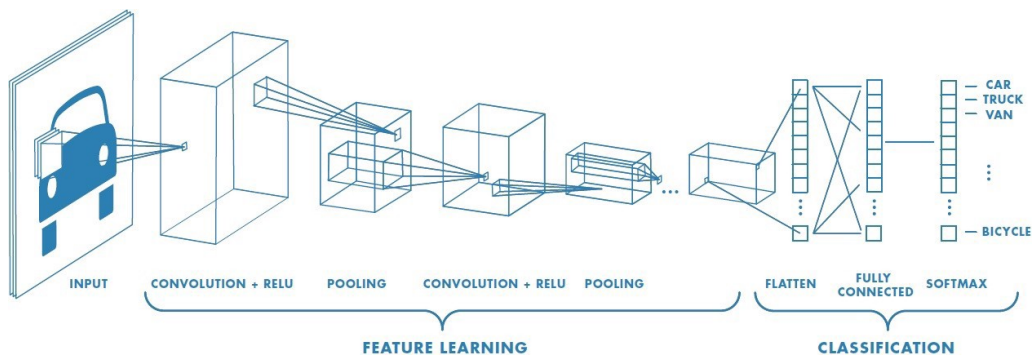


Figure 2.1 - Structure of a convolution neural network[6].

In general, a standard convolutional neural network has four elements: convolution kernel, pool sampling, activation function, and full-connection. The convolution kernel is applied to extract higher-level abstraction of the input image, namely the feature map. Pooling sampling aims at reducing the feature map size. The activation function is to provide nonlinearity in the neural network classification. The fully connected layer is used as the classification output to predict the final results. Finally, the backpropagation algorithm is applied in a CNN to train the entire neural network. A structure of a convolution neural network is shown in the figure 2.1.

2.1.1 Network layers

2.1.1.1 Convolution (Conv)

The convolution usually implements an operation similar to filtering. It applies a $K \times K$ size convolution kernel to the input feature map to perform vector point multiplication shown in the figure 2.2.

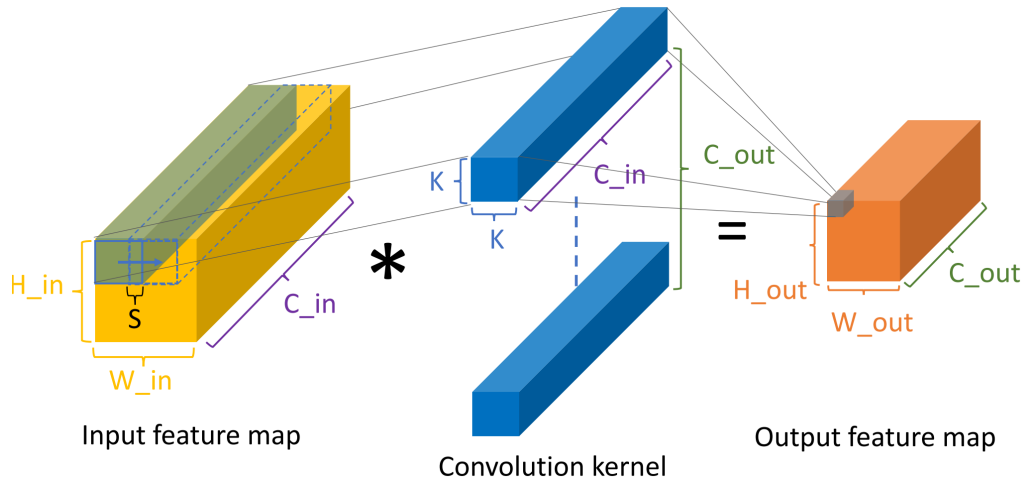


Figure 2.2 - Example of the convolution operation.

The heavy computation of CNN is almost the convolution operation[109], we thus make a deep study to explain the process of the convolution clearly. Table 2.1.1.1 lists all necessary parameters in a convolution operation. The convolution input is an input feature map with $W_{in} \times H_{in} \times C_{in}$ and the kernel with $K \times K \times C_{in} \times C_{out}$. The convolution output is the feature map with the dimension $W_{out} \times H_{out} \times C_{in} \times C_{out}$. The convolution operation is essentially accumable multiple and addition. After filling the input feature maps with the parameter P , the convolution operation firstly intercepts the feature map with $K \times K \times C_{in}$ dimension on the filled input feature map. This feature

map will be multiplied and accumulated with the three-dimensional convolution kernel of the same size to obtain a pixel of the output feature map. Secondly, the operation will traverse the input feature maps according to the stride S to obtain a two-dimension $W_{out} \times H_{out}$ result. Repeating these two steps in the C_{out} channel, a final feature map with $W_{out} \times H_{out} \times C_{out}$ can be conducted. Finally, a bias is added. The expression of the output \mathbf{O} is 2.1:

$$\mathbf{O}[n][x][y] = \mathbf{B}[n] + \sum_{k=0}^{C_{in}-1} \sum_{i=0}^{W_{in}-1} \sum_{j=0}^{H_{in}-1} \mathbf{I}[k][Sx+i][Sy+j] \times \mathbf{W}[n][k][i][j] \quad (2.1)$$

$0 \leq n < C_{out}, 0 \leq x < W_{out}, 0 \leq y < H_{out}$

Parameter	Detail
W/H_{in}	Width and height of the input image
W/H_{out}	Width and height of the output image (feature map)
C_{in}	Number of input channels
C_{out}	Number of output channels
K	Kernel size
S	Stride of the convolution
P	Padding of the convolution

2.1.1.2 Sampling (Pool)

Sampling is a process of downsampling, aiming to reduce image size and computation while keeping necessary information. Two types of sampling have been used in the CNN execution: maximum pooling and average pooling. During the sampling, the number of the channel remains unchanged. Taking max-pooling operation as an example, the feature map is usually divided into multiple large and small rectangular areas. Then, the maximum value in each sub-area is selected for output. For example, the expression of max-pooling 3x3 output \mathbf{O} can be described as 2.2:

$$\mathbf{O}[n][x][y] = \max(\mathbf{I}[k][Sx][Sy], \mathbf{I}[k][Sx][Sy+1], \mathbf{I}[k][Sx][Sy+2], \mathbf{I}[k][Sx+1][Sy], \mathbf{I}[k][Sx+1][Sy+1], \mathbf{I}[k][Sx+1][Sy+2], \mathbf{I}[k][Sx+2][Sy], \mathbf{I}[k][Sx+2][Sy+1], \mathbf{I}[k][Sx+2][Sy+2]), \quad (2.2)$$

$0 \leq n < C_{out}, 0 \leq x < W_{out}, 0 \leq y < H_{out}$

2.1.1.3 Activation

The activation function constructs a key part of CNN, which provides the nonlinear factor in the neural network to solve the linear indivisible problems in CNN. There are some common-used activation functions:

- Sigmoid: Sigmoid is a widely used function. The value range of output is $[0,1]$, since it is often used as an output function, and the output value is expressed as a probability.

The expression is:

$$f(u) = \frac{1}{1 + e^{-u}} \quad (2.3)$$

- ReLU: ReLU is the abbreviation of Rectified Linear Unit, and its characteristics are close to biological nerve, its expression is:

$$f(u) = \max(0, u) \quad (2.4)$$

- tanh: Tanh is a traditional activation function, which is obtained by dividing the hyperbolic sine \sinh and the hyperbolic cosine \cosh . It can also be regarded as a variant of the Sigmoid function, with a value range of $[-1,1]$ between. Its expression is:

$$f(u) = \tanh(u) \quad (2.5)$$

2.1.1.4 Fully-connected (FC)

The fully connected are stacked behind the convolutional layer to finally extract the results. All the inputs from the previous layers are connected to every activation unit of the next layer. A fully connected layer applies a filter to the input feature map like a convolutional layer, while the kernel size is the same as the input feature map. Therefore, let $W_{in} = K$, $W_{out} = 1$, and $S = 1$ in the equation 2.6, we can obtain the equation of the fully-connected:

$$O[n] = B[n] + \sum_{k=0}^{C_{in}-1} \sum_{i=0}^{W_{in}-1} \sum_{j=0}^{H_{in}-1} I[k][i][j] \times W[n][k][i][j] \quad (2.6)$$

$0 \leq n < C_{out}$

2.1.2 Inference and training

CNN requires training and inference. First, a set of data is collected and labeled as a training data set to train CNN in the context of supervised training. Then, when

the training gradually converges with the expected results, we can use the data set for inference to obtain the accuracy.

2.1.2.1 Training

Training is a process of learning to find weights in convolution and fully connected layers, which can minimize differences between output predictions and given ground-truth labels on a training dataset [159] through several epochs. To train a neural network, we should define some hyperparameters to control the training process, such as learning rate, mini-batch size, number of epochs, momentum.

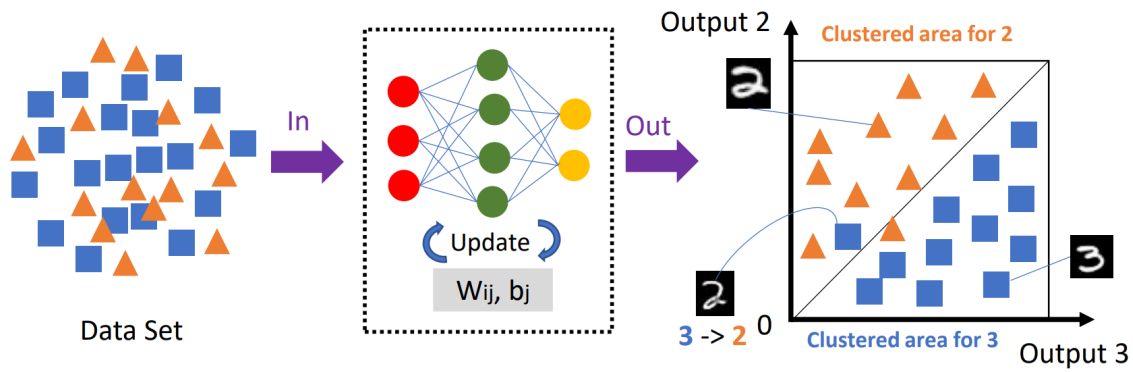


Figure 2.3 - Example of the training process

In training, most of the training of the neural network is based on the backpropagation algorithm. Then, the weight and bias are updated by a gradient descent optimization algorithm.

The backpropagation consists of the following steps:

- Forward the network model to get the activation of each layer including the output layer;
- Calculate the derivative of the activation function and residual of each layer;
- Calculate the partial derivatives of the cost function concerning the weights W and the biases b respectively;
- Update the W and b .

If the dataset of the network is m , then the cost function $J(W,b)$ of an l -layer network can be expressed as:

$$\begin{aligned}
J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{j=1}^{S_l} \sum_{i=1}^{S_{l+1}} (W_{ji}^{(l)})^2 \\
&= \left[\frac{1}{m} \sum_{i=1}^m \left(\frac{1}{2} \|h_{W,b}(x^{(i)}) - y^{(i)}\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{j=1}^{S_l} \sum_{i=1}^{S_{l+1}} (W_{ji}^{(l)})^2
\end{aligned} \tag{2.7}$$

s_l represents the number of activation of the l -th layer. Likewise, n_l represents the number of layers of the network model.

The gradient descent method can be used in training to continuously adjust the weight W and the bias b according to a particular learning rate α , and obtain the optimal solution of W and b .

$$\begin{aligned}
W_{ji}^{(l)} &= W_{ji}^{(l)} - \alpha \frac{\partial}{\partial W_{ji}^{(l)}} J(W, b) \\
b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)
\end{aligned} \tag{2.8}$$

The partial derivatives of the cost function for w and b can be expressed as:

$$\begin{aligned}
\frac{\partial}{\partial W_{ji}^{(l)}} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial W_{ji}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda (W_{ji}^{(l)}) \\
\frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)})
\end{aligned} \tag{2.9}$$

The residual $\delta_i^{(l)}$ can be expressed as:

$$\delta_i^{(l)} = \sum_{j=1}^{S_{l+1}} \delta_j^{(l+1)} \cdot W_{ij}^{(l)} \cdot f'(z_i^{(l)}) \tag{2.10}$$

Therefore, the derivation formula of the cost function expressed by the residual error concerning the parameters W and b can be described as:

$$\begin{aligned}
\frac{\partial}{\partial W_{ji}^{(l)}} J(W, b) &= \frac{\partial J(W, b)}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial W_{ji}^{(l)}} = \delta_i^{(l+1)} \cdot a_j^{(l)} \\
\frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{\partial J(W, b)}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}} = \delta_i^{(l+1)}
\end{aligned} \tag{2.11}$$

2.1.2.2 Inference

The inference applies a pre-trained model from a training phase to infer the results of the dataset. When a dataset is sent to the pre-trained network, it outputs a prediction

based on the predictive accuracy of the neural network. Differs from the training phase, inference will not update the weights and biases of layers according to the predicted result if the results have errors.

2.2 FIELD PROGRAMMABLE DEVICES

Field Programmable Gate Array (FPGA) is a programmable logic array that can be re-configured after production. The basic structure of FPGA includes programmable input and output units (IOBs), configurable logic blocks (CLBs), digital clock management modules, embedded block RAM, wiring resources, embedded dedicated hard cores, etc. Moreover, it consists of several Digital Signal Processor (DSP) which can process the special high-precision operation under maximum 500MHz[82]. FPGA logics are realized by loading programming data into the internal static storage unit. The value stored in the storage unit determines the function of the logic unit. It also determines the connection between the modules and the connection between the modules and I/O.

Later, a system-on-chip (SoC) FPGA with an integrated processor appeared, such as the Xilinx Zynq-7000 series [155]. The processing system (PS) side is usually dual-core, and the runtime control logic is executed on the Linux operating system or bare system. On the other hand, the programmable logic (PL) side has traditional FPGA resources such as DSP and LUT and can be configured for various applications. The communication between PS and PL is realized through different intellectual property (IP) and AMBA AXI.

2.3 DESIGN FLOW OF APPLICATION OF FPGA

The FPGA design flow can be described as the following figure 2.5, from specifying the design with constraints to finally implementing the design on FPGA.

2.3.1 *System specification*

"System specification" describes the specifications and constraints of the system, such as FPGA type, system function, required clock frequency, etc. In this step, the hardware designer determines the software and hardware parts for Intellectual Property (IP) design and the interconnection standards between the IP and the processor. At the same time, hardware designers can determine the necessary hardware resources to respond to previously defined constraints. You can also decide whether you need to design a verification process to ensure that the system is working correctly.

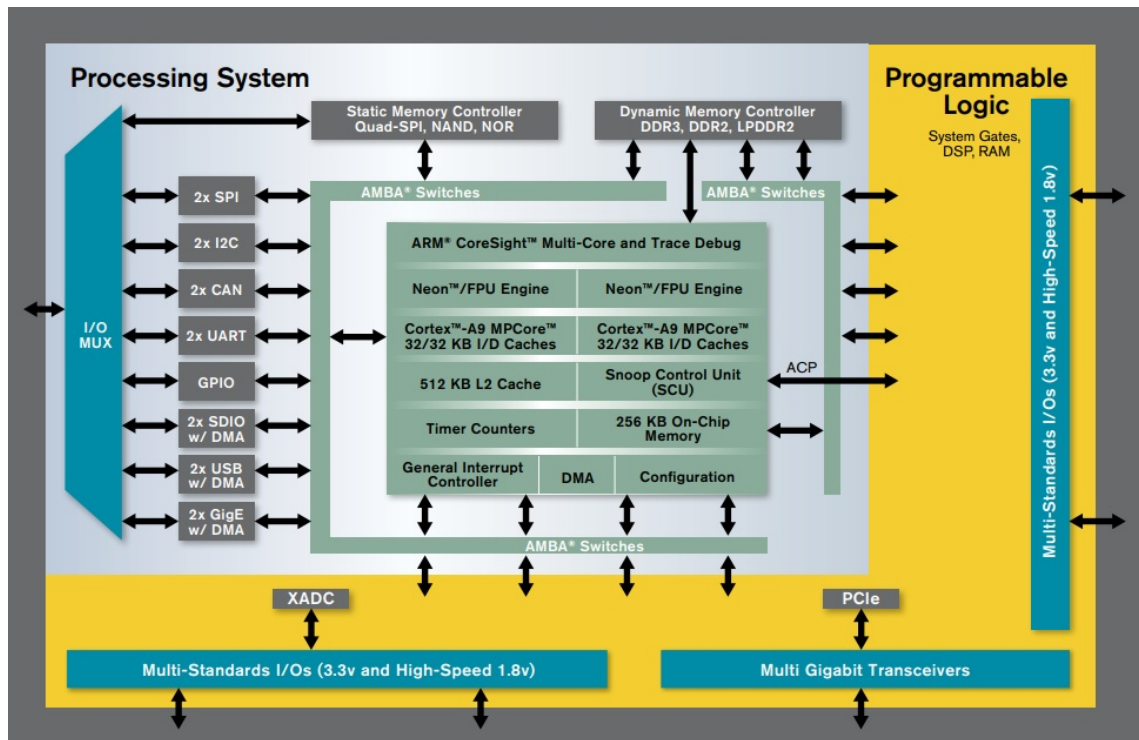


Figure 2.4 - Xilinx Zynq-7000 block diagram[155].

2.3.2 IP design

"IP design" is to realize the functions of IP. Once the characteristics of IP have been identified in the step "System specification", the hardware designer can choose several methods to describe the design.

The traditional method uses hardware description language (HDL) such as VHDL and Verilog to generate register transfer level (RTL), which requires solid hardware acknowledges. The high-level synthesis (HLS) approach has come into being, allowing the bitstream or the RTL code from the software language, such as C++/C. This approach is more productive with less implementation time but at the cost of efficiency. For example, [107] proves that Key-value Store (KVS) implementation time in HSL has been reduced by 20% compared to HDL language flow.

It is also possible to use open source IPs or commercial IPs provided by the FPGA companies (e.g., Xilinx, Altera) or IP providers in this step.

2.3.3 IP integration

"IP integration" is to interconnect all the IPs to achieve a complete system on the FPGA. It is possible to apply some data transfer protocols among the IPs. The most popular

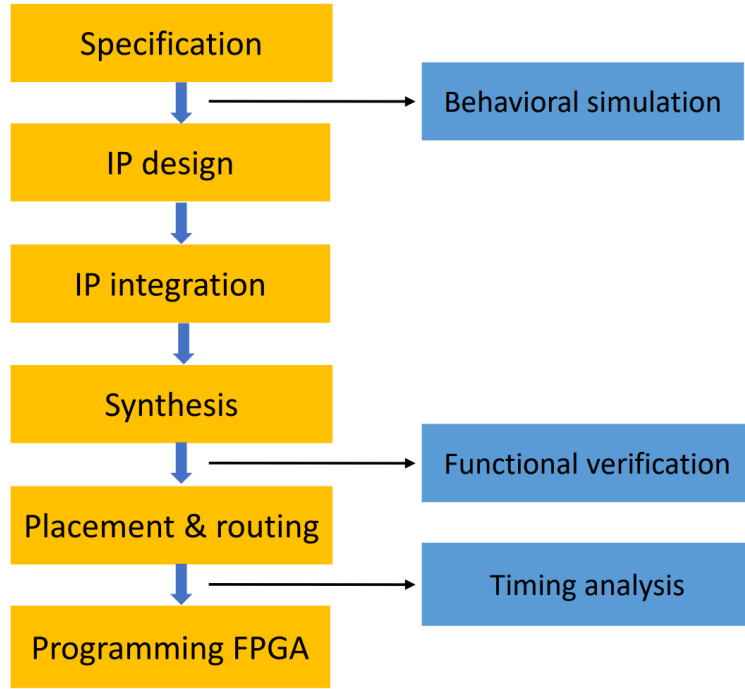


Figure 2.5 - FPGA design flow.

standard is AMBA Advanced eXtensible Interface 4 (AXI4), which is supported by Xilinx [154] and Altera [15]. However, Altera also has its standard, Avalon [14]. In addition, it also exists several open-source standards, such as Wishbone [7]. These standards have several types of interconnections, which are summarised in the table 2.1. The type of the interconnection are following:

- Point-to-point type, usually used for the pipeline;
- Bus type, which allows one or more master devices to be connected to multiple masters to multiple slaves;
- Network on chip (NoC), which allows multiple IPs to be connected for parallel communication.

Table 2.1 - Standards and interconnections between IPs.

Standard	Point to point	Bus	Others
AMBA	AXI streaming	AXI MM, AXI lite, AHB, APB	
Avalon	Avalon-ST	Avalon-MM	Avalon conduit, Avalon-TC
Wishbone	Point to point	Shared bus, crossbar	

In our work, we use the standard AXI4 for the IP interacting with the processor of the FPGA. There exist three types of AXI4: AXI4-full, AXI4-lite, AXI4-stream.

The AXI4-full is a high-performance memory-mapped data and address interface that contributes to burst access to memory-mapped devices. The AXI4-lite is a subtype of AXI4, which has a more straightforward interface compared to AXI4-full. However, AXI-lite has not the burst access capability, which is suitable for a lightweight data transfer. The AXI4-Stream protocol is usually used for data-centric applications or the data flow paradigm, in which the address of data is not necessary for the design. Each AXI4-Stream acts as a single unidirectional channel for the handshake data stream [152]. Table 2.2 lists several features of the AXI standard.

Table 2.2 - Difference between AXI4-full, AXI4-lite, AXI4-streaming.

Interface	Features	Data type
AXI4	Traditional memory mapped address/data interface	Data burst supported
AXI4-Lite	Traditional memory mapped address/data interface	Single data cycle only
AXI4-Stream	Only data-interface	Data-only burst

2.3.4 Synthesis

"Synthesis" is to convert the hardware architecture written in VHDL or Verilog into a netlist, which is a netlist representing electrical diagrams. This step can reveal some errors that were not detected during the functional simulation process, for example, timing closure issues that would lead to functional degradation in the system.

The synthesis can be carried out with the tools of the FPGA supplier or by third-party software. It is necessary to simulate the system after the synthesis. This simulation-based on the netlist is more precise than the functional simulation carried out before synthesis.

2.3.5 Placement & routing

"Placement & routing" is to implement the architecture on a specific FPGA. During the implementation step, the netlist is attached to a particular FPGA. Next, the netlist components are placed on the internal structures of the FPGA, such as BRAMs, IOs, and registers. Then, these resources are routed while respecting the constraints of the FPGA. It is possible to add placement and timing constraints when performing this step. A timing analysis file is available and is generated. This file makes it possible to make a simulation integrating the propagation times. In addition, it makes it possible to analyze the causes of non-compliance with the design constraints.

2.3.6 Programming FPGA

"Programing FPGA" is to download the bitstream generated in the design to the FPGA. Bitstream is a binary file that can configure FPGA. After obtaining the binary file, you can configure the FPGA again to change the design. You can also generate a binary file in the EEPROM so that the FPGA configuration information will not be lost after the power is turned off.

In this step, if the design has some bugs, you can use several debugging tools provided by FPGA manufacturers. For example, Xilinx offers an integrated logic analyzer (ILA), an oscilloscope that allows the observation of signals inside the FPGA. In addition, to debug the soft code of the CPU, Xilinx provides software called Xilinx SDK to verify the function of the CPU code. In addition, you can check the bus (AXI4) function to verify the data conversion between PS and PL. For example, Xilinx has AXI Verification IP (AXI-VIP), written in SystemVerilog, and uses Universal Verification Method (UVM) to implement software/hardware co-simulation.

2.4 CHALLENGES OF CNNs ON FPGAs

Figure 2.6 exhibits a general method of performing CNN training and inference phases. Under the condition of resources and bandwidth, the research on accelerating CNN inference on the FPGA has attracted more and more attention. Due to the complexity of the backpropagation algorithm and the need for high-precision data, the training phase is usually completed in a CPU- or GPU-based software platform. First, the network model with configurations is used on software and hardware platforms. Then, the training phase is based on the collected floating-pointed training dataset and test dataset. In several training iterations, the network will output the pre-training floating-pointed weights and biases. Finally, these floating-pointed data will be quantized into the fixed point format on the CPU- or GPU-based platform for the inference stage.

In the inference stage, as shown in figure 2.6, the FPGA platform has a network generation. Network generation completes several calculations in CNN or the entire CNN. The generator obtains the network configuration from the training phase, reads the input data from the outside, and generates approximate output for the CNN. In the process of execution, the network generation may be used multiple times.

The increasing scale and the innovating structures of CNN's bring challenges to realize

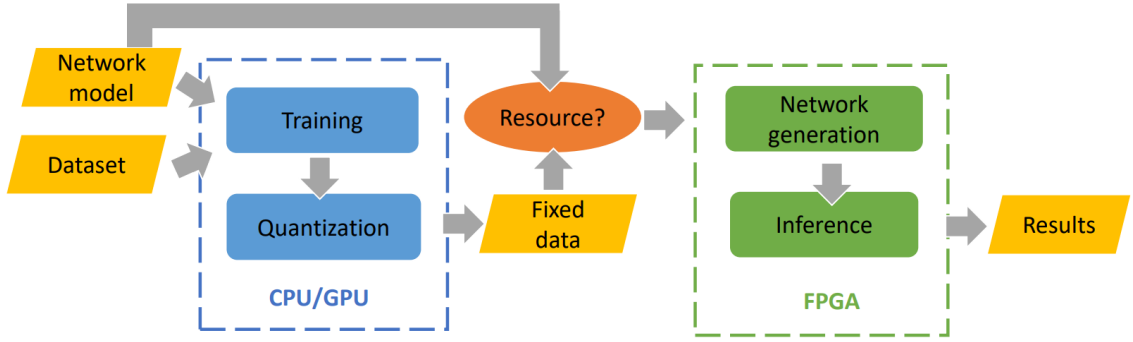


Figure 2.6 - General method for the CNN.

the CNN implementations on the FPGA. For the power consumption, the high-speed data flow inside the system, such as the data flow between the storage and the network computation block occupies a part of power consumption. For the throughput performance, the hardware resources and memory bandwidth are the key elements that limit the performance of the FPGA-based acceleration module factors. Based on the structures of the CNNs, how to improve performance and efficiency in the FPGA-based accelerators, reducing resource and power consumption, are needs to be further explored.

The challenges can be expanded as follows:

- **Resource and bandwidth limitations:** The most advanced neural networks usually have a large number of deep layers of computational operations, such as AlexNet (8 layers, 724M MAC) [81], GoogleNet (22 layers, 1.6G MAC) [60] and ResNet (layer 152, 11.3G MAC)[60], which may cause insufficient resources for CNN deployment on the FPGA. Besides, in traditional CNNs, the convolution layer takes up a large percentage of computation and the data transmission (for example, AlexNet and VGG-16 accounts for 90% of the total computation [58]), which increase the off-chip bandwidth demand from weight transfer for large CNNs.
- **Network generality:** CNN structure has different convolutional layers and kernel sizes. General hardware modules that can adapt to other network structures should be designed to promote the deployment of CNN on FPGAs. Moreover, frameworks that automatically accomplish the CNN structure generation and implementation process are also desired.

ACCELERATING CNNs FROM LOCAL TO VIRTUALIZED FPGA IN THE CLOUD: A SURVEY OF TRENDS

Contents

3.1	INTRODUCTION	22
3.2	BACKGROUND	26
3.2.1	FPGA cloud	26
3.2.2	FPGA virtualization	27
3.3	CNN IMPLEMENTATION TECHNIQUES	30
3.3.1	Hardware architecture design	30
3.3.2	Network compression	31
3.3.3	Optimization strategy	32
3.4	ACCELERATING CNNs FROM LOCAL TO VIRTUALIZED FPGAs IN THE CLOUD	33
3.4.1	CNNs on local FPGA	33
3.4.2	CNNs on local virtualized FPGA	36
3.4.3	CNNs on virtualized FPGA in the cloud	38
3.4.4	CNN deployment in commercial cloud	40
3.5	TRENDS OF CNN ACCELERATORS	40
3.6	DISCUSSION	45
3.6.1	Unresolved challenges	45
3.6.2	Industrial solution	46
3.6.3	Roadblocks of FPGA Cloud	47
3.7	CONCLUSION	47

Field-programmable gate arrays (FPGAs) are widely used locally to speed up neural network algorithms (e.g., CNN) with high computational throughput and energy efficiency. Virtualizing FPGA and deploying FPGAs in the cloud are becoming increasingly attractive methods for network acceleration because they can enhance the computing ability to achieve on-demand acceleration across multiple users. In the past five years, researchers have extensively investigated various directions of FPGA-based CNN accelerators, such as algorithm optimization, architecture exploration, capacity improvement, resource sharing, and cloud construction. However, previous CNN accelerator surveys mainly focused on optimizing the CNN performance on a local FPGA, ignoring the trend of placing CNN accelerators in the cloud's FPGA.

In this chapter, we conducted an in-depth investigation of the technologies used in FPGA-based CNN accelerators, including but not limited to architectural design, optimization strategies, virtualization technologies, and cloud services. Additionally, we studied the evolution of CNN accelerators, e.g., from a single CNN to framework-generated CNNs, from physical to virtualized FPGAs, from local to the cloud, and from single-user to multi-tenant. We also identified significant obstacles for CNN acceleration in the cloud. This chapter enhances the current understanding of the evolution of FPGA-based CNN accelerators.

3.1 INTRODUCTION

Neural networks have become a cutting-edge research topic owing to their excellent performance in image classification, detection, segmentation, and data prediction. Owing to the remarkable prediction capacity of datasets in a wide range of complex applications, researchers have proposed myriad networks, such as AlexNet [81], VGG16 [127], ResNet152 [61], Transformers [139], General Adversarial Networks (GANs) [47], and Variational Autoencoder (VAE) [78]. The success of CNNs has also attracted attention in the development of industrial platforms, such as Google Deepmind [106], Facebook AI [59], Amazon Alexa [89].

Traditionally, in academia and industry, graphics processing units (GPUs) are used to train CNNs, as they provide a high degree of parallelism to process these algorithms [130, 105]. However, the execution of CNNs on GPU-based platforms encounters energy/power and throughput bottlenecks. In 2016, a tensor processing unit (TPU) was announced by Google [74], which runs CNNs 15 to 30 times faster than contemporary GPUs using

similar technologies [73], and the energy efficiency is increased by a factor of 30–80. Despite its speedup and energy efficiency, the TPU has a high production cost, lacks reconfigurability, and cannot be adapted to the emergence of new network models with complex structures.

Field-programmable gate arrays (FPGAs) can achieve energy efficiency and high performance in the face of rapidly innovating CNN models and computational characteristics, as reported by Venieris et al [144]. FPGAs can achieve up to 20 tera multiply accumulates per second (TMACs), and the power consumption does not exceed 25 W, incurring a less than 10% overhead in the overall power consumption [4]. Moreover, FPGAs can provide a flexible hardware architecture with a fine granularity and massive pipeline level. Therefore, FPGAs have become an alternative method for accelerating CNNs.

Early CNN accelerators (e.g., [183, 132, 117, 37]) are typically implemented on a single local FPGA fabric. As the number of learnable parameters and operations in CNNs increases, the resources of a single FPGA may be insufficient for the entire CNN deployment. The challenges of designing CNNs on a single local FPGA are described below.

- **Productivity:** Owing to the complexity of CNN design, mapping a CNN onto an FPGA requires specific hardware expertise in hardware description language programming and performance optimization, which have long learning curves. According to the complexity of the CNN algorithm, deploying the CNN on the FPGA may be time-consuming and may increase the programming burden of designers. In recent years, productivity has improved owing to the emergence of compilation frameworks that automatically map CNNs onto the FPGA.
- **Scalability:** CNNs are computation- and data-intensive applications that require enormous computational resources. For example, VGG-16 has up to 39 billion operations and more than 500 million parameters for 224×224 image classification [83]. In deeper CNNs, the resource requirements may exceed the available resources in a single FPGA, limiting the scalability of the CNN architecture. Even if technologies and strategies are adopted to optimize the CNN architecture, when a large-scale CNN is deployed in a single local FPGA, the resource bottleneck can easily be reached.
- **Elasticity:** The solution of deploying CNN accelerators on local FPGAs lacks re-

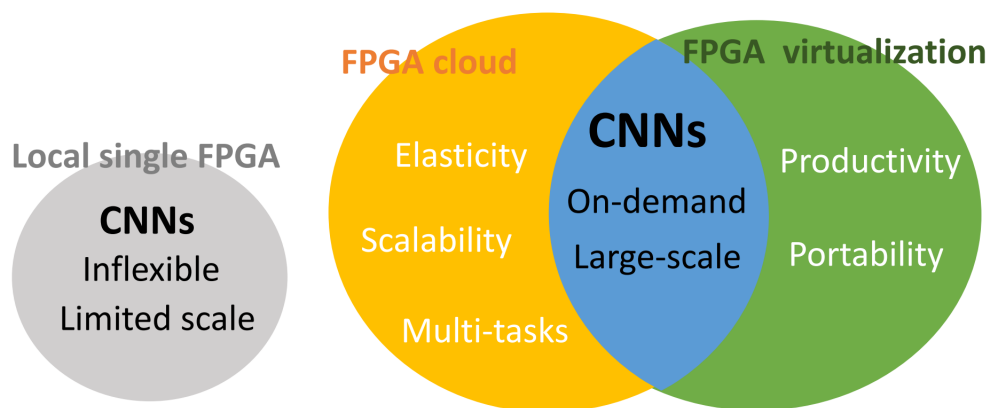


Figure 3.1 - Characteristics of deploying FPGAs in the cloud and FPGA virtualization for CNN deployment.

source elasticity because it assumes that CNN resource allocation must be fixed throughout the deployment lifecycle. Because different CNN algorithms require different computing resources, memory bandwidths, and storage resources [175], these solutions cannot flexibly provide and deprovision resources at runtime and hence fail to match different workloads of the CNN.

- **Portability:** The deployment of most CNN accelerators directly depends on the characteristics of the FPGA platform and is therefore restricted to a specific FPGA vendor. Owing to the lack of an abstraction layer that isolates CNN accelerators from specific FPGA platforms, these accelerators may face portability issues of CNN structures. They cannot adapt quickly to the current changing CNN algorithms.
- **Multi-tasks:** Generally, the execution mode of a CNN on a local FPGA is limited to a single user executing a single CNN within a given time. It remains difficult for a single local FPGA to support multiple users by executing execute multiple CNNs in parallel and satisfy each user's time, cost, and quality of service (QoS) requirements. Some frameworks (for example, [33]) successfully solve the problem of multiple CNN scheduling but can only execute CNNs sequentially in the form of time slices in a single-task environment.

Deploying FPGAs in the cloud and/or virtualizing FPGAs can resolve the aforementioned challenges, as shown in Figure 3.1. The cloud paradigm enhances the computing capability of FPGAs with high throughput and low latency. It enables the sharing of single or multiple FPGA resources across multiple users, thereby efficiently scaling and accelerating on-demand CNNs. Virtualizing FPGAs abstracts low-level physical resources and

hides hardware design and compilation flow from the software designers' view, providing a high-level application-dependent architecture. Owing to FPGA virtualization, software designers can deploy CNN accelerators according to different requirements (e.g., throughput, execution time, and accuracy) without relying on a specific FPGA platform. The collaboration between FPGA virtualization and the FPGA cloud can satisfy the resource requirement, thereby taking advantage of the "unlimited" cloud capability to flexibly scale CNN accelerators.

While exploring techniques to accelerate CNNs on local physical FPGAs, researchers have also attempted to adopt FPGA virtualization and the FPGA cloud to facilitate the implementation of multiple CNNs at a large scale and achieve flexible deployment in a multi-user environment. Although the FPGA cloud and virtualization have brought breakthroughs to the deployment of CNNs, previous surveys (e.g., [101, 53, 22, 87, 23, 103, 36]) have mainly focused on CNN optimization and the design of local FPGAs (e.g., architecture design, simplification, optimization strategies). These surveys ignore the trend of CNN implementation on the timeline, that is, from local to virtual FPGA in the cloud. Moreover, no in-depth analysis or comparison of the challenges faced by the CNN accelerators at different stages of deployment was conducted. Relying on previous surveys, we aim to

- Provide an overview of the main techniques of FPGA-based CNN accelerators. These techniques were initially proposed to optimize the performance of CNN accelerators in a local FPGA, but they can also be applied to the FPGA cloud environment.
- Present the evolution of CNN accelerator deployment from local to virtualized FPGAs through an in-depth introduction of virtualization techniques and the FPGA cloud.
- Perform an in-depth analysis and comparison of the challenges faced by the CNN accelerators at each stage.

The chapter is organised as follows: Section 3.2 provides an FPGA cloud definition and a general overview of FPGA virtualization. Section 3.3 discusses the crucial approaches for accelerating CNNs on the FPGA, which is also applicable to the FPGA cloud. Section 3.4 describes the use of virtualization technology in local CNNs and cloud-based CNNs. Section 3.5 highlights the trends and evolution of the FPGA-based CNN and compares

the characteristics of these accelerators. Section 3.6 discusses the unresolved challenges of accelerating CNNs in the FPGA cloud and presents other directions for CNN acceleration. Section 3.7 gives the conclusion of the survey.

3.2 BACKGROUND

This section presents an overview of the FPGA cloud and the available services in the cloud and introduces the FPGA virtualization technology from the viewpoints of the abstraction level and system architecture.

3.2.1 *FPGA cloud*

Deploying FPGAs in the cloud involves leasing a bundle of specific software tools, platforms, or FPGA resources remotely in a cost-effective manner. Such an FPGA-enabled cloud maintains the advantages of FPGAs (e.g., low power consumption and programmability) and establishes scalability, elasticity, and multi-tenancy.

Provisioning FPGA resources is similar to provisioning traditional central processing unit (CPU)- and GPU-based clouds. Regarding the service categories in traditional cloud computing, FPGA cloud providers offer FPGAs as infrastructure as a service (IaaS) or software as a service (SaaS) [116]. Figure 3.2 presents an example of hierarchical mapping in the FPGA cloud. There is no standard definition or classification for FPGA clouds, and the hierarchical mapping may change over time.

3.2.1.1 *FPGA in IaaS*

The FPGA in IaaS provides access to the FPGA computing resource pool and memory storage in the cloud. This paradigm divides the FPGA into multiple independent virtual instances and supports high-bandwidth communication to collaborate between each resource instance. Per-FPGA or multiple-FPGA granularity can be supported in the IaaS for application deployment. Cloud users must manually map their applications to resources if their applications are deployed across multiple FPGAs.

As a commercial example, the Amazon F1 instance offers a collection of eight FPGA devices with a high bandwidth. Enabling FPGA in IaaS has also attracted attention in the academic field. Byma et al. [26] abstracted FPGAs into virtual regions and managed resources across multiple FPGAs through OpenStack. Asiatici et al. [19] provided a

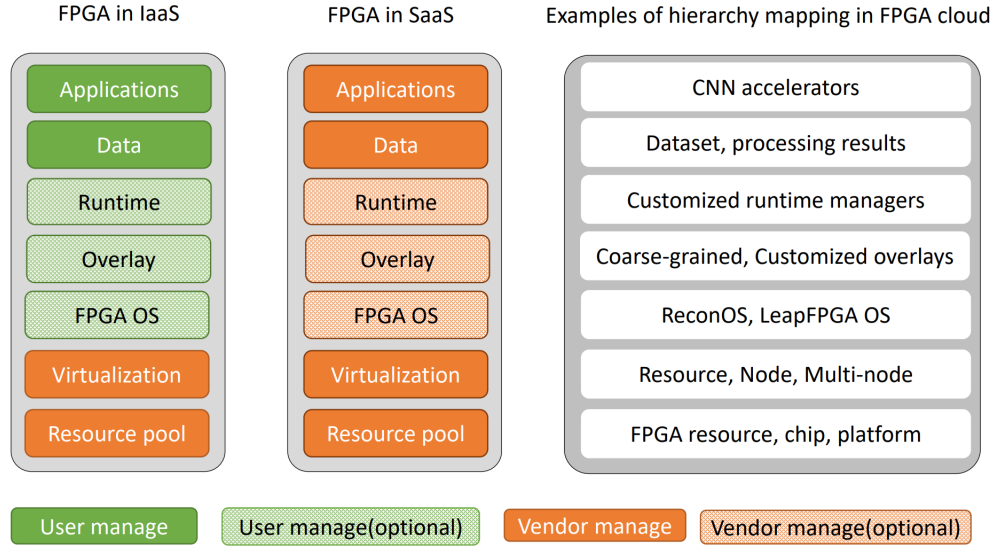


Figure 3.2 - IaaS and SaaS FPGA cloud. "Vendor manage (optional)" and "User manage (optional)" indicate that this hierarchy does not always exist in the FPGA cloud, and it is customised by each FPGA cloud vendor or user.

runtime management framework to map FPGA resources for different applications with limited overhead.

3.2.1.2 FPGA in SaaS

The FPGA in SaaS offers acceleration services for cloud users to execute applications and process data. Technical processes have been hidden in the cloud background, and cloud users do not need to be responsible for the hardware design flow and FPGA resource management. For example, Microsoft released the Catapult project [108], which puts Altera Stratix vFPGA per CPU in the cloud to accelerate the Bing web search engine, with a 95% improvement throughout. Moreover, Microsoft released the BrainWare project, where FPGAs are used to accelerate state-of-the-art CNNs in major services such as Bing and Azure [30].

3.2.2 FPGA virtualization

The objectives of FPGA virtualization are to 1) provide a virtual abstraction of resources and underly the low-level hardware design from users; 2) support FPGA sharing in the time and space domains to serve multiple tasks; and 3) facilitate the hardware design process and accelerate the program compilation [67, 138, 145, 128]. We review FPGA virtualization according to the abstraction level [137] and system architecture [111]. The definition of FPGA virtualization has changed over time in different scenarios.

3.2.2.1 Abstraction level

According to the scale of resource computing, FPGA virtualization can be divided into three abstraction levels: resource, node, and multi-node levels.

- **Resource level:** The resource level contains reconfigurable resources (e.g., logic) and non-reconfigurable resources (e.g., Input/Output blocks). Several uniform architectures, such as coarse-grained overlays, have been proposed to support the portability of this level between different types of FPGAs [79, 148].
- **Node level:** The node level considers a single FPGA as a node. Resource allocation and scheduling are concerned with a single FPGA at this level. Currently, time-division multiplexing (TDM) and space-division multiplexing (SDM) are the two principal methods for sharing a single FPGA resource [85, 168].
- **Multi-node level:** The multi-node level is designed to assign resources in multiple FPGAs to multiple applications or multiple users. However, mainstream compilation tools only support application deployments on a single FPGA [157]. Therefore, application mapping across FPGAs requires specific frameworks to solve hardware problems, such as intercommunication, resource partitioning, and traversing the physical boundary.

3.2.2.2 System architecture

The system architecture refers to a structural view at the abstraction level. It usually covers the hardware, software stack, and overlay [129] but may be different at each level of abstraction. Here, we introduce the system architecture in a node-level abstract form, as shown in Figure 3.3, which can also be applied to other levels of abstraction.

- **Hardware stack:** The hardware stack can vary in the host interface, shell, and role.
 - **Host interfaces:** 1) on-chip host inside the FPGA, which can be a soft core formed by programmable logic (PL) or a hard core in the processing system (PS) of a system-on-a-chip (SoC) FPGA; 2) local host, local CPU host, connected via high-bandwidth links (e.g., PCIe); 3) remote host placed remotely via the network.
 - **Shell:** The shell is a static region, usually comprising a system memory controller (e.g., DRAM adapter), interface controller (e.g., DMA controller), and network

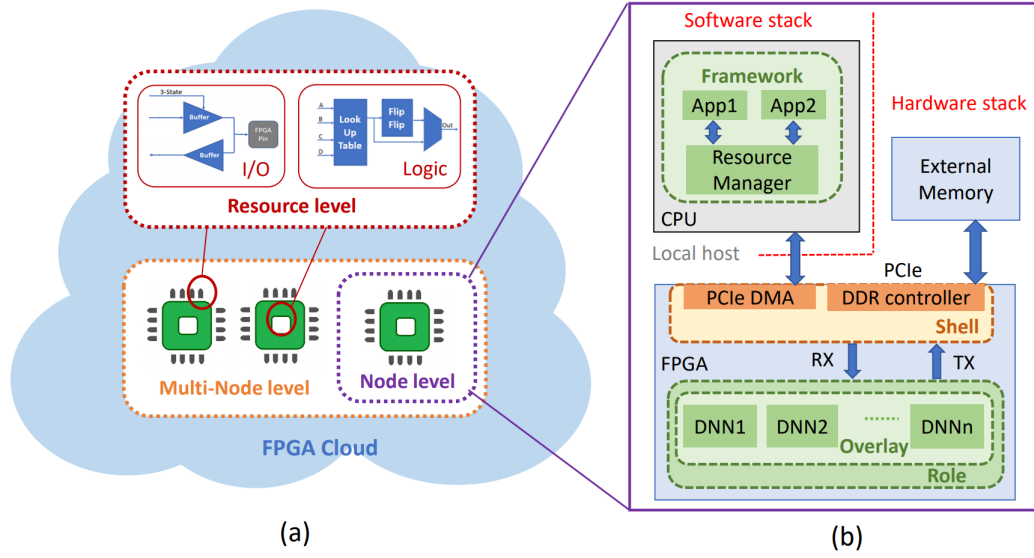


Figure 3.3 - Overall architecture of the FPGA-based CNN accelerators in the IaaS cloud. (a) Different levels of abstraction in the FPGA virtualization technique. (b) Example of the system architecture in node-level virtualization.

interface controller (e.g., Ethernet core). For instance, the shell in [91] includes the user PCIe, management PCIe, card management system, and DDR access channel.

- Role: The role is a dynamic region in the FPGA, which can be regarded as a reserved region for deploying CNNs in our context. It runs independently of the shell and can be reconfigured every time for each application to satisfy user requirements.
- Software stack: The software stack runs on a host, provides users with an application programming interface, and enables the communication between the host and the FPGA. [111] introduces three types of software stacks: 1) Operating systems (e.g., LeapFPGA OS [38], Recon OS [11]), which are conceived to support multiple threads for runtime resource management. 2) The host application, which is written in OpenCL and C++, provides simultaneous access to a shared FPGA for multiple users. 3) Software frameworks (e.g., OpenStack), which can be used to share resources across multiple users and distribute several partial reconfigurations to one FPGA.
- Overlay: The overlay provides an intermediate layer between the hardware stack and the software stack to achieve program portability. It is considered a virtual reconfigurable architecture on top of a physical FPGA. Fine-grained granularity and coarse-grained granularity in overlays are used in various applications [28, 86].

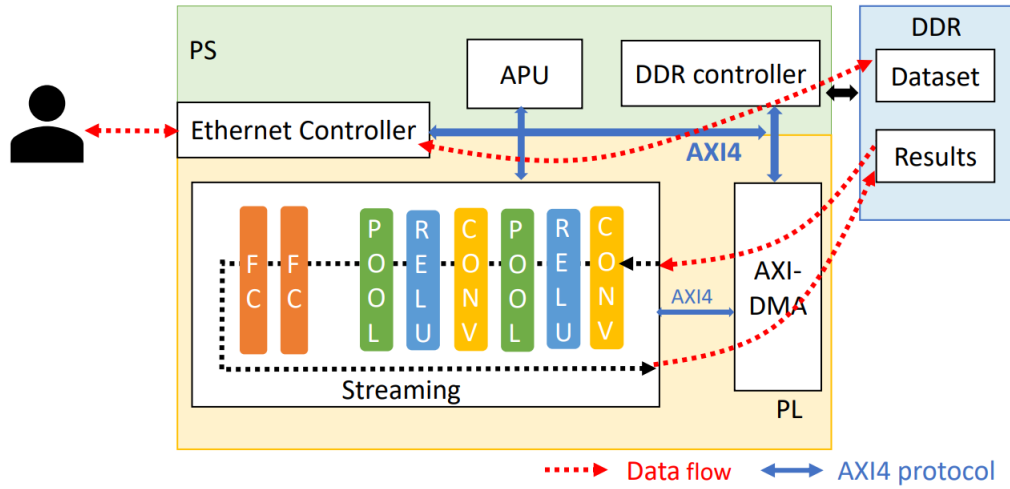


Figure 3.4 - Example of accelerating CNNs using the streaming architecture.

3.3 CNN IMPLEMENTATION TECHNIQUES

To enhance the performance of CNNs on the FPGA locally and in the cloud, several techniques have been extensively studied. This section presents implementation techniques that have been recently investigated.

3.3.1 Hardware architecture design

The widely used hardware architecture are streaming and single computation engine architectures.

- **Streaming architecture:** The streaming architecture (Figure 3.4) implements an entire CNN on the side of the Programmable logic (PL) of the FPGA from the first convolutional layer to the final fully connected layer. On the PL side, it deploys a chain of sequential CNN intellectual property (IP) to process the dataset in the pipeline mode. The intermediate results (CNN feature maps) are stored on the chip on the PL side. This architecture enables an efficient data stream without frequent data exchange with external memory, significantly reducing the latency and obtaining throughput at a high frequency. A specific CNN model using a streaming architecture must be defined before generating the bitstream. Whenever the CNN model changes, architecture re-compilation and bitstream regeneration are inevitable. According to the selected CNN algorithm, the re-compilation of this architecture may be time- and resource-consuming.
- **Single Computation Engine:** Single computation engine (Figure 3.5) implements a

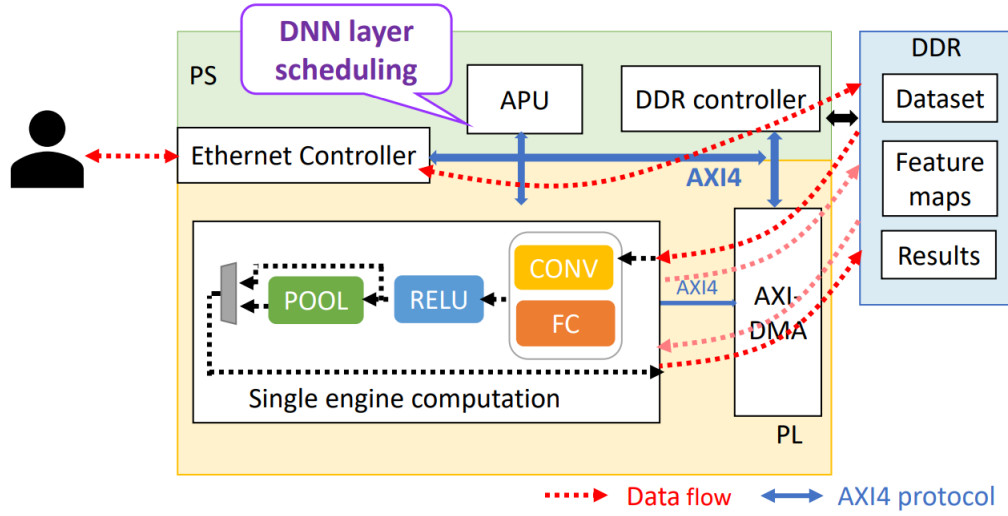


Figure 3.5 - Example of accelerating CNNs using the single computation engine accelerator architecture.

part or a layer of the CNN on the PL side. It is a universal fixed template, usually in the representation of a systolic array or multiple processing elements that can be configured as CNN layers of different scales [143, 110]. The execution of the entire CNN is achieved sequentially by configuring this template in the program on the PS side. The intermediate results (CNN feature maps) are stored off the chip. The architecture significantly reduces resource usage and introduces possibilities to scale accelerators. It has been widely used in accelerators to enrich CNN diversity. However, because the CNN blocks are executed sequentially on the FPGA, the execution time is extended significantly. Each time the CNN architectures changes, it is necessary to reload a complete bitstream to realize the novel CNN deployment on the FPGA.

Table 3.1 presents the major features of streaming and single computing engine architectures according to their performance (e.g., flexibility, reconfiguration, resource consumption).

3.3.2 Network compression

The increasing amounts of learnable parameters and arithmetic operations of CNNs lead to a computational burden and additional resource consumption of hardware devices. Network compression makes CNNs more compact when the data width is limited, assisting in striking a balance between resource usage and accuracy. Thus far, quantization, pruning, and in-parallel pruning quantization have been successfully employed for

Table 3.1 - Comparison of streaming and single computation architectures for CNN acceleration.

	Streaming	Single computation
Network implementation	Entire network	Function unit
Structure	Pipeline	Recurrent
Optimization mode	Layer-independent	One-optimization-fit-all
Recompilation time	Long	Short
Reconfiguration	Bitstream-level reconfiguration	Processor control configuration
Flexibility	Low	High
Resource usage	High	Low
Speedup	Fast	Low

network compression.

- **Quantization:** Network quantization converts floating-point data to fixed-point data with a selectable data width. Quantization includes uniform quantization with the same width for all network layers or dynamic quantization of each layer based on the layer characteristics. Researchers have widely adopted 16-bit fixed-point quantization (for example, [50, 150]), and 4- and 8-bit uniform quantization [56, 90] have already achieved good accuracy. Therefore, uniform quantization of a small width is promising owing to its ease of implementation on FPGA while maintaining accuracy.
- **Pruning:** Network pruning removes nonsignificant neurons to avoid overfitting. This is an efficient method, particularly in embedded systems, for reducing the network size and saving computing resources to fit the network to the memory size [102]. In [176], the authors compressed a trained CNN model and performed reverse pruning and peak pruning with fewer weights. Compared with the GPU, the compressed AlexNet on FPGA achieved 182.3× and 1.1× improvements in latency and throughput, respectively.

3.3.3 Optimization strategy

The scale of complex CNN structures introduces resource challenges. Moreover, the data (e.g., weights) stored in the external memory require enormous energy and latency. Because CNNs are composed of massive repeated loop operations, unrolling and tiling can be used to weaken off-chip communication and deal with parallel computation problems. A more detailed optimization was presented in [101].

- **Loop unrolling:** Unrolling executes a network or multiple layers in parallel—particularly

convolutional layers. The network can be fully expanded to achieve massively parallel processing or apply appropriate unrolling factors (iterations in the loop) across different layers for partial unrolling in the for-loop to optimize the datapath and maximise the throughput [52, 96]. Ma et al. [94] adopted four types of loop unrolling in kernel maps and feature maps to determine the parallelism scheme and maximise data reuse. In an experiment involving VGG-16 on an Arria 10 FPGA, a throughput of 645.25 GOPS was achieved.

- **Loop tiling:** Constrained by limited on-chip memories, the data to be processed are tiled into multiple tiles and stored in on-chip buffers. Selecting a suitable tiling size factor can determine the trade-off between resources and the required external memory bandwidth. For example, Yu et al. [93] designed an auto-compilation process based on RTL, which uses intra-block and inter-block strategies to divide the layer execution into multiple sequential tiles. The process designed in [146] supports both unrolling and tiling of input and output feature maps on binarised networks. A 2× area efficiency improvement was achieved compared with existing binarised networks.

3.4 ACCELERATING CNNs FROM LOCAL TO VIRTUALIZED FPGAs IN THE CLOUD

The work of accelerating CNNs on FPGAs in our surveys covers local to the cloud and integrates the virtualization technique. The metrics used to evaluate these methods usually include throughput, power, and accuracy. Additionally, the adoption of virtualization techniques introduces additional characteristics such as portability and productivity, and in the cloud environment, QoS and isolation are regarded as new characteristics.

3.4.1 CNNs on local FPGA

Early studies (e.g., [84, 10, 35]) were dedicated to manually mapping a CNN model to a local FPGA with a streaming architecture. These studies take full advantage of CNNs parallelism and apply layer-independent optimization strategies to fit the entire network into the FPGA.

Benefiting from the well-defined structure of modern CNNs, which contain similar layers with repetitive operations, researchers have proposed frameworks with a single-engine computation structure [96, 174, 50, 123, 123, 158, 65], as shown in Figure 3.6. These frameworks take advantage of both software programmability and flexible hardware

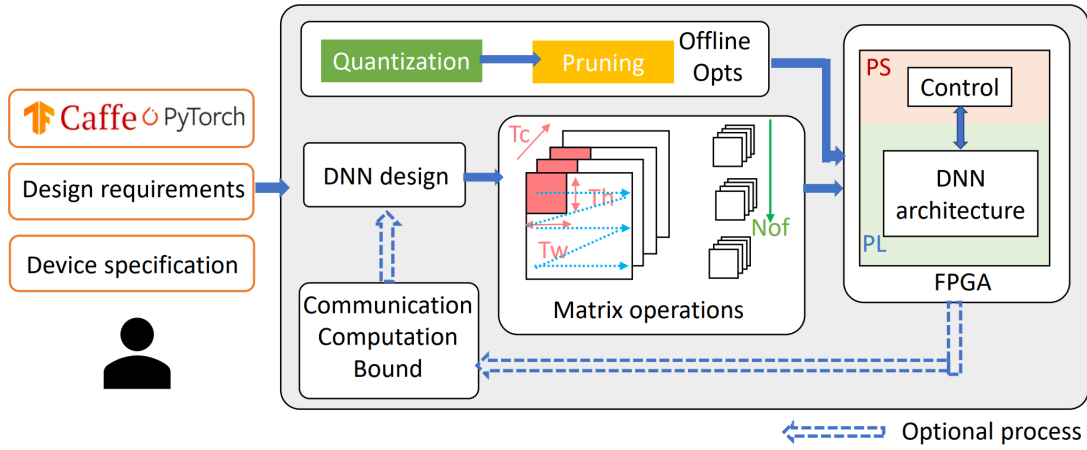


Figure 3.6 - Generic frameworks for CNN accelerators.

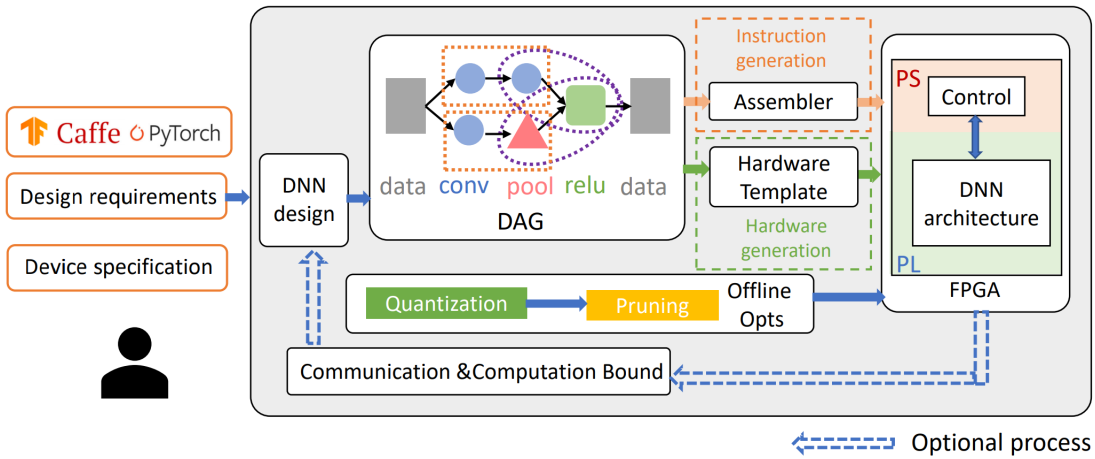


Figure 3.7 - Generic compiler-inspired frameworks for CNN accelerators.

structures, making CNN implementation more diversified and achieving high performance with reduced resource consumption. More frameworks that automatically map single CNNs to local FPGAs were presented in [143]. Another new type of framework is a toolchain that includes a compiler [52, 156, 9]. The compiler is a CNN architecture-aware tool that can map a wide range of CNN applications to the instruction set architecture (ISA) and control signals [52]. Figure 3.7 presents an example of a compiler-inspired toolchain. Wang et al. [156] proposed a compiler that transforms a CNN deployment into a graph-level problem. The compiler first takes the software description as input and then transforms the description into directed acyclic graphs of computational operations. The networks generated by the compiler on Xilinx ZU9 reach throughputs of 2.82 TOPs/s (VGG), 1.38 TOPs/s (ResNet50), and 1.41 TOPs/s (GoogleNet).

More works are presented in Table 3.2.

Table 3.2 - Several examples of manual mapping and frameworks on the local FPGA.

	Works	Year	Models	Device	Data format	Architecture	Strategy	Perform. (GOPs)
Manual	[171]	2015	Costum CNN	Xilinx Virtex vc707	32-bit FP*	Single engine	Unrolling	61.62
	[54]	2015	CIFAR10	Xilinx Kintex 325 T	16-bit	Single engine	Quantization	260
	[110]	2016	VGG-16	Xilinx Zynq XC7Z045	16-bit	Single engine	Unrolling, tilling	137.3
	[112]	2016	AlexNet	Xilinx vc707	32-bit FP*	Single engine	Unrolling, tilling	75.16
	[84]	2016	AlexNet	xilinx vc709	16-bit	Streaming	Quantization, ping-pong buffer, batching	565.94
	[179]	2017	LRCN	Xilinx vc710	16-bit	Single engine	Pruning, quantization, unrolling, tilling	75.5
Framework	[52]	2017	VGG16	Xilinx Zynq xc7z020	8-bit	Single engine	Per-layer quantization	84.3
			YOLO					62.9
			LeNet-5					185.81
	[140]	2016	MPCNN	Xilinx Zynq xc7z02	32-bit FP*	Streaming	Pipeline	100.23
			CNP					150.91
			CFF					159.22
	[136]	2017	BNN-SFC	Xilinx Zynq ZC706	1-bit	Streaming	Pipeline, binarized network	8265
			BNN-LFC					908
			BNN-CNV					246
	[50]	2017	VGG-19	Altera StratixV SG5MD5	16-bit	Single engine	Tilling, batching	364.36
			LSTM-LM					315.85
			REsNet-152					226.47
	[46]	2017	AlexNet	Xilinx Zynq XC7Z045	16-bit	Single engine	Quantization, loop removal, rearrangement	120.3
			GoogLeNet					116
			ResNet-50					122.3
	[174]	2018	AlexNet	Xilinx UltraScale KU060	16-bit	Single engine	Unrolling, pipeline	163
			VGG16	Xilinx Virtex vc709				354
				Xilinx UltraScale KU060				266
	[96]	2018	AlexNet	Altera Stratix V GXA7	8-bit	Single engine	Unrolling	114.5
			NiN					117.3
			VGG					334
	[156]	2019	ResNet50	Xilinx ZU2	8-bit	Streaming	Quantization, tilling	228.7
			GoogLeNet					231.5

* FP = Floating point format.

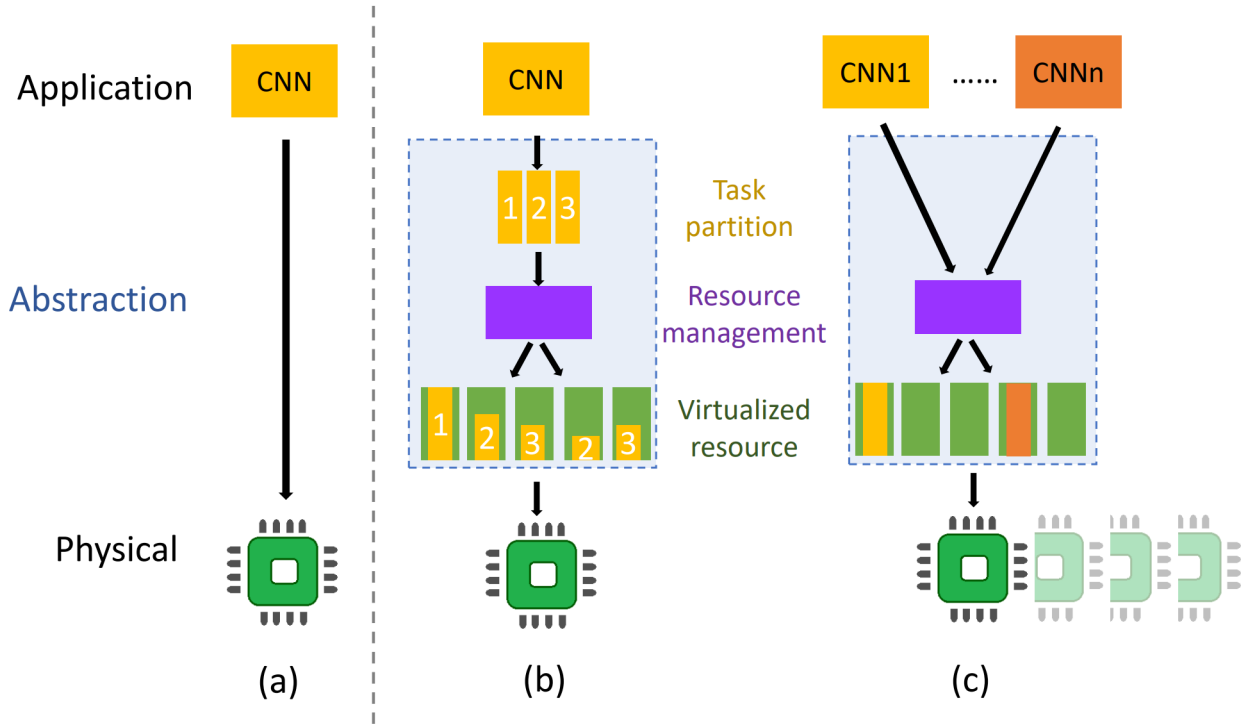


Figure 3.8 - (a) CNN deployment without virtualization. (b) Example of FPGA virtualization at the node level for deploying one CNN on a local FPGA. (c) Example of FPGA virtualization at the node level for deploying several CNNs in the cloud environment.

3.4.2 CNNs on local virtualized FPGA

FPGA virtualization bridges the gap between the hardware stack and the software stack with the abstraction layer, enhancing the productivity and portability of CNN applications. Virtualization also enables resource sharing among multiple FPGAs with flexible resource management to support a wide range of CNNs. Figure 3.8 shows an example of virtualization at the node level.

At the resource level of virtualization, Tong et al. [44] propose a coarse-grained overlay-based framework for quantising and accelerating a CNN with any data width on an FPGA. The coarse-grained array comprises a reconfigurable NoC, a scheduler, and network computation components and is configured as CNN models according to instructions generated by the compiler of the framework. Such an overlay is independent of FPGA features and can be flexibly adapted to FPGAs provided by different vendors. To satisfy the metrics in virtualization, e.g., reducing the time and complexity of CNN reconfiguration, this framework reconfigures the coarse-grained array from the rightmost column to the leftmost column. The results indicate that the inference of AlexNet and VGG-16 on Xilinx UltraScale+ VCU118 takes only 0.13 and 2.63 ms, respectively.

Table 3.3 - Comparison of coarse- and fine-grained overlays on FPGAs for CNN acceleration

	Coarse-grained overlay	Fine-grained overlay
Logic level	RT level	Gate level
Data width	Up to 32bit	1 bit
Example logic	CNN function unit (e.g., Conv)	Control instruction (e.g., Load)
Goals	Optimise CNN datapath switch	Enable CNN variability
Advantage	Area-efficiency	Higher flexibility

Similarly, Struharik et al. [131] designed a coarse-grained overlay-based accelerator consisting of a set of processing blocks, which enabled on-the-fly reconfiguration for different CNNs. The accelerator can implement mainstream CNN families, such as VGG, Inception, ResNet, MobileNet, and NASNet, with a frame rate up to 6.05 times higher than that of Nullhop [12]. Other methods [57, 18] also employ a coarse-grained overlay on top of the FPGA to enable dynamic datapath reconfiguration of CNN applications at runtime.

In contrast to previous studies where CNNs were deployed on FPGAs using the coarse-grained overlay, several researchers adopted a fine-grained overlay as an abstraction level to achieve higher flexibility. Venieris et al. [142] proposed an automated framework for implementing multiple CNNs on a target FPGA platform with fast space exploration. The framework adopts a streaming architecture to allocate resources at a fine-grained granularity for exploring a wide range of resource and bandwidth allocations. The authors tested their framework in a multi-CNN system (ZFNet, VGG16, SceneLabelCNN) on Xilinx ZC706, and the results indicated that the framework achieved an improvement of up to $6.8\times$ in performance/W over Nvidia Tegra X1. Table 3.3 presents the features of the overlays used in the previous studies.

In node-level virtualization, the resource of a single FPGA can be allocated to a single CNN application or multiple CNN applications in TDM or SDM [92]. Zhang et al. [180] developed an end-to-end framework called a DNN builder to build CNNs with high performance using a design space exploration strategy. The DNN builder enables virtualization on a single physical FPGA by allocating resources to several small accelerating engines. The resource allocator can generate parallel schemes and data buffering guidelines for each layer. The tool deploys AlexNet, ZF, VGG16, and YOLO on Xilinx XC7Z045 and KU115 and achieves up to $5.15\times$ better performance than that reported in [166].

At a multi-node level, allocating resources from multiple FPGAs to the CNN appli-

cation may result in performance degradation owing to insufficient off-chip bandwidth. Therefore, it is essential to employ optimized resource mapping and efficient communication for this virtualization level. Zhang et al. [177] enabled large-scale CNN application implementation across up to 16 FPGAs with resource- and bandwidth-aware mapping methods. Taking the FPGA topology, resource conditions, and neural-network specifications as the inputs, this method can partition the CNN application to each FPGA depending on the statuses of the FPGAs (busy or free) and the estimation throughput of layer mapping. Results indicated that ResNet-152 on a multi-FPGA architecture outperformed a single-FPGA deployment by a factor of 16.4. Geng et al. [42] developed a framework that adopts a pipelined architecture to train CNNs on multiple FPGAs with a one-dimensional topology. The pipelined architecture with the fine-grained inter- and intra-layer methodology minimises the time required for storing the feature map in the memory during training. The authors evaluated their framework by training AlexNet on 10 Xilinx VC709 Connectivity Kits. The results indicated that compared with other frameworks [175], the throughput obtained by this framework was increased by a factor of 5; compared with Titan X, the energy efficiency of the framework was up to 7.6 times higher. Moreover, the framework exhibits good scalability, as it can scale up to 60 FPGAs to accelerate CNNs.

However, such multiple-FPGA platforms adopting pipeline models gain high throughput while sacrificing latency. Jiang et al. [71] developed a general framework called Super-LIP to support concurrent processing for both single- and multi-layer deployment on FPGAs. To achieve communication between two FPGAs, the authors employ a novel methodology in Super-LIP to achieve linear speedup by balancing computation workloads and distributing the shared data across FPGAs to avoid traffic heaviness on the FPGA memory bus. Compared with the existing single-FPGA design [171], this method achieved a 3.48 \times speedup of AlexNet, VGG, and YOLO on two Xilinx ZCU102 kits.

3.4.3 CNNs on virtualized FPGA in the cloud

Zeng et al. [168] proposed a framework using FPGA virtualization, which is applicable to any CNN accelerator based on the ISA in a cloud environment. This principle divides a large resource pool into multiple virtualized cores to share FPGA resources at the node level. By introducing a novel two-level instruction (dispatch module and tiling-based instruction package design), virtualized multi-core resources can be dynamically allocated

to each block in one CNN (single-task mode) or each CNN for multiple users (multi-task mode) at runtime. Compared with previous methods, this technique solves physical resource isolation and performance among multiple users by sharing FPGA resources in the SDM method. Experiments on VGG-16, ResNet50, Inception V3, and MobileNet indicated that compared with a single non-virtualized core design, the throughput of the proposed virtualization method with multiple cores was 1.07–1.69 times higher overall.

A similar method called ViTAL was developed by Zha et al. [169] to enable FPGA virtualization in a cloud environment for deploying CNNs. This method supports resource sharing at both the node and multi-node levels. ViTAL provides an abstraction layer between CNN applications and physical resources, which abstracts heterogeneous resources into homogeneous resources and provides a view of virtual blocks. The abstraction layer divides a CNN application into virtual blocks and then maps these virtual blocks to an FPGA or multiple FPGAs without impacting other running CNNs. By using a latency-insensitive interface, virtual blocks can be mapped across FPGAs at the multi-node level to achieve timing closure and match communication delays. Additionally, isolation in the cloud environment is achieved by avoiding the sharing of physical resources among different virtual blocks. The authors evaluated ViTAL by implementing LeNet, AlexNet, and VGG-16 on a Xilinx UltraScale+ FPGA. The experimental results indicated that ViTAL achieved good CNN mapping quality with a short compilation time (1.6% of the total). Furthermore, ViTAL can dynamically relocate the CNNs to different positions in the FPGA. The experimental results also indicated that with FPGA virtualization methods, ViTAL significantly shortened the response time (by 82%) in the cloud environment.

Fowers et al. [40] proposed a full-system architecture with virtualization at a multi-node level to serve CNN inferences in a cloud environment. The critical feature of the architecture is the dedicated neural processing units (NPU), which implement an SIMD ISA containing a matrix-vector multiplier. This CNN-specific ISA offers a high-level abstraction between the underlying FPGA infrastructure and DNN software development, thereby simplifying FPGA programming for software developers. The authors validated the architecture by running RNNs and compared it with the NVIDIA Titan GPU, and it gained more than 36 effective teraflops (10 instances NPU). Moreover, the authors evaluated ResNet-50 on the Arria 10 GX 1150, which achieved 559 inferences per second (IPS), whereas ResNet-50 on the Nvidia P40 GPU achieved only 461 IPS.

3.4.4 CNN deployment in commercial cloud

In recent years, companies such as Amazon F1 [1], Tencent [133], Huawei FACS [2], and Microsoft [5] have launched cloud projects that provide FPGA IaaS for users to rent FPGA resources. Researchers have begun to accelerate CNN workloads in these commercial clouds to improve performance. The framework of deploying CNNs on FPGAs with a commercial cloud as the backend is similar to local deployment, but virtualization and physical connections of FPGAs are often charged by cloud vendors and hidden in the backend.

Several frameworks [113, 27, 135] have been proposed to implement CNNs on a single physical FPGA in the cloud with Caffe and TensorFlow as a frontend. Later, the research focus of CNN deployments moved from per-FPGA granularity to multiple FPGAs. Because the mainstream compilation tools do not support application implementation among multiple FPGAs, particular mapping algorithms or customized tools designed by the researchers are needed. Shan et al. [121] proposed an effective solution for implementing CNNs among multiple FPGAs in an AWS instance. The solution, which is based on the characteristics of FPGAs in the AWS, uses a heuristic method to find the global execution throughput between the CPU and the connected FPGA and then uses an allocation algorithm (including group kernel allocation and individual kernel allocation) to assign CNN workloads to various FPGAs with resource constraints. It is suitable for deploying any CNN to the AWS F1. Compared with the traditional mixed-integer nonlinear programming solution, this solution achieved faster CNN implementations on multiple FPGAs: 16-bit fixed-point AlexNet on two FPGAs, 32-bit floating-point AlexNet on four FPGAs, 16-bit fixed-point VGG-16 on four or six FPGAs, and ResNet on five FPGAs.

Table 3.4 presents studies on virtualization technology and the cloud environment.

3.5 TRENDS OF CNN ACCELERATORS

As shown in Figure 3.9, the first stage in the evolution of CNN accelerators involved manually mapping a single CNN to a single local FPGA with low energy consumption. CNN accelerators were designed for implementation on specific FPGA families. The optimization strategies are customised for a particular CNN and are not compatible with other networks. Therefore, CNN deployment has disadvantages, such as poor portability,

Table 3.4 - Several examples of CNNs based on local virtualized FPGAs and CNNs in the cloud.

Works	Years	CNN Model	Abstraction	Host	Shell	Cloud	Multi-tenant	Task and rsc. Controller	FPGA Device	Num.
[175]	2016	AlexNet, VGG-16	Multi-node level	RH	AXI, Network access	N/A	N/A	system controller	Xilinx Virtex VC709	6
[30]	2018	LSTM, RNN	Multi-node level	RH	Network communication, PCIe controller	N/A	N/A	Model parallelism, On-chip pinning	Altera Stratix 10 280	1
[42]	2018	AlexNet	Multi-node level	LH	Communication, I/O component	N/A	N/A	Partitioning, Memory subsystem	Xilinx Virtex VC709	10
[72]	2018	VGG-16, AlexNet, SqueezeNet, YOLO	Multi-node level	LH	Communication, PCIe, Aurora	N/A	N/A	Mixed integer linear programming	Xilinx XC7Z015 /XC7Z045 /X-CZU9EG	1
[149]	2019	AlexNet, Disp-Net, ResNet, GoogLe-Net	Resource level	OC	AXI, Memory controller	N/A	N/A	Tuning algorithm	Xilinx ZC706, ZCU102	1
[71]	2019	AlexNet, Squeeze-Net, YOLO, VGG-16	Multi-node level	OC	Host communication, clock generator	N/A	N/A	Hypervisor	Xilinx ZCU102	2
[177]	2019	ResNet-152	Multi-Node level	RH	Network communication	N/A	N/A	Dynamic partitioning	Xilinx UltraScale	16
[125]	2019	Customized 3D CNN	Multi-Node level	LH	Network interface, PCIe controller	N/A	N/A	Hardware monitor, mapping table	Xilinx VCU118	4
[62]	2019	Squeeze-Net, GoogLe-Net, VGG-16	Resource level	LH	DDR controller, Global memory interconnection	N/A	N/A	Partial reconfiguration manager	Intel Stratix 10 SX SoC	1

Continued on the next table

Continued from previous table

Works	Years	CNN Model	Abstraction	Host	Shell	Cloud	Multi-tenant	Task and rsc. Controller	FPGA Device	Num.
[44]	2020	VGG-16, AlexNet	Resource level	OC	DDR controller	N/A	N/A	Croase-grained NoC, Parameter scheduler	Xilinx VCU118	1
[75]	2018	DNN Weaver	Node level	LH	PCIe controller, DMA, MMIO	Cloud environment	N/M	Zone manager on host CPU	Altera Stratix V GS, Xilinx Ultrascale+	1
[30]	2018	LSTM, RNN	Multi-node level	RH	Network communication, PCIe controller	Cloud environment	SM, TM	Resource runtime manager	Altera Stratix 10 280	1
[134]	2020	CIFAR-Net	Node level	OC	Memory controller, AXI	Cloud environment	SM, TM	Runtime task manager, scheduling decision	Xilinx ZCU104	1
[169]	2020	NiN, AlexNet, OverFeat, Vgg-16	Multi-node level	RH	Latency-intensive interface, address translation	Cloud environment	TM	Hypervisor and system controller	Xilinx UltraScale+	4
[168]	2020	ResNet-50, Inception V3, MobileNet	Node level	LH	Virtualization infrastructure	Cloud environment	SM	Multi-level instructions, virtualization manager	Xilinx Alveo U200	1
[50]	2017	VGG-19, ResNet-152, LSTM	N/M, Charged by cloud vendors			IaaS (Per-FPGA)	N/A	Symbolic compiler	Catapult	1
[24]	2018	MLP, YOLO, DoReFa-Net	N/M, Charged by cloud vendors			IaaS (Per-FPGA)	N/A	Data Flow Balancing algorithm	AWS F1	1
[113]	2018	LeNet, VGG-16	N/M, Charged by cloud vendors			IaaS (Per-FPGA)	N/A	Datamover, system controller	AWS F1	1

Continued on the next table

Continued from previous table

Works	Years	CNN Model	Abstraction	Host	Shell	Cloud	Multi-tenant	Task and rsc. Controller	FPGA Device	Num.
[27]	2019	AlexNet, VGG-16, ResNet-50	N/M, Charged by cloud vendors			IaaS (Per-FPGA)	N/A	Model split, task allocation manager	AWS F1	1
[120]	2020	AlexNet, VGG-16	N/M, Charged by cloud vendors			IaaS (multi-FPGAs)	N/A	MINLP Solver	AWS F1 and F2	1
[21]	2015	DenseNet-121, ResNet-152, etc		N/M		SaaS (Per-FPGA)	SM, TM	Web service API	Arria10	1, 2, 4

N/A = Not applied; N/M = Not mentioned; SM = Spatial multiplexing; TM = Time multiplexing.

time-consuming deployment, complex optimization, and inflexibility. Efforts have been made to automatically generate CNN hardware structures according to the requirements of different FPGA families.

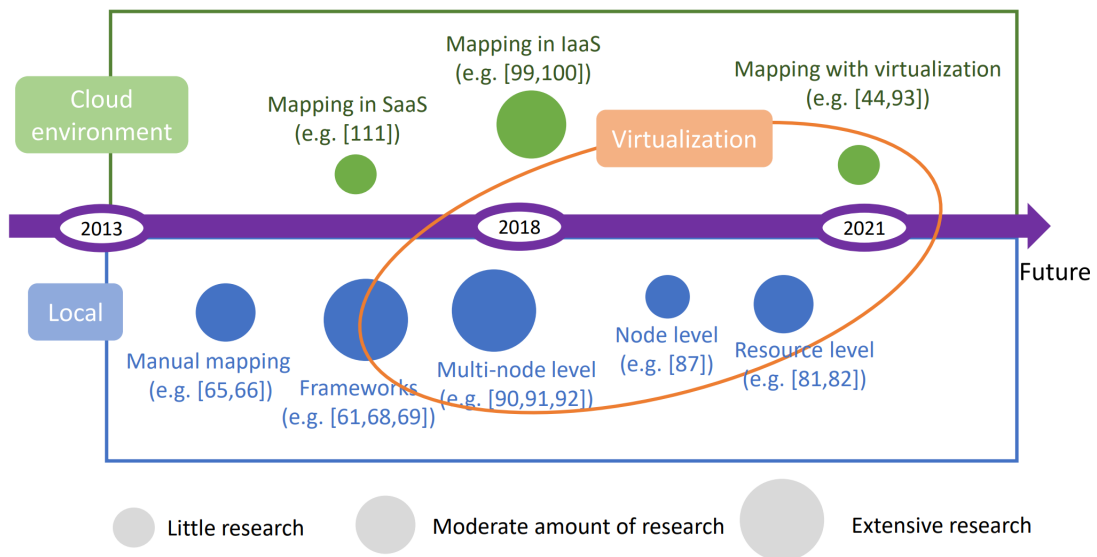


Figure 3.9 - Evolution of CNN accelerators at each time node: from manual mapping to frameworks, from a single node to a cluster, from physical to virtual resources, and from local to cloud.

Therefore, researchers have proposed several frameworks to support a generic CNN accelerator and to offer customised CNN implementations by analysing requirements and platform-specific constraints. These frameworks are usually integrated with an RTL compiler with full exploitation of low-level structures to achieve high performance. Moreover, instruction-driven compiler frameworks have been developed in recent years to simplify the control flow of CNNs.

Despite enjoying energy efficiency and acceleration, CNN deployment on FPGAs faces complexity, resulting in reduced productivity. The framework of the previous stage mainly reduces the programmable complexity at the single-FPGA level without multiple tasks, and researchers have not yet determined how to improve the productivity of CNN implementation at the multiple-FPGA or resource level. Accordingly, CNN accelerators with virtualization techniques are being developed. Resource-level virtualization provides portability of CNN deployment for various families of FPGAs from different vendors. Node-level and multi-node-level virtualization enables resource sharing among FPGAs. Multi-node level virtualization exhibits the advantages of scaling up CNNs and training CNNs.

Subsequently, several works proposed cloud-based accelerators for deploying CNNs on-demand. These studies can be divided into two categories. The first category involves building an end-to-end cloud environment for CNN acceleration. These works not only require the development of a framework or a solution for CNN deployment but are also responsible for providing FPGA devices, virtualizing FPGAs, managing FPGA resources, scheduling tasks, and supporting multi-tenant scenarios with resources and data isolation. However, these works are still in their infancy and face obstacles, such as runtime overhead. Few researchers have performed studies in this area, but it will be an appealing field owing to the growing focus on cloud computing. The other category involves using the commercial FPGA cloud as a backend to develop CNN frameworks or solutions. These frameworks usually cannot consider multi-tenant solutions and cannot support CNN deployment at runtime. Additionally, FPGA management and virtualization are handled by the cloud provider and hidden in the background. Studies have mainly focused on deploying CNNs at per-FPGA granularity because this does not require resource-mapping algorithms or compilation tools across multiple FPGAs. CNN development can only be completed by using cloud integrators provided by cloud vendors and mainstream compilation tools.

At each stage, CNN deployment exhibits various characteristics, as shown in Figure 3.10. Most CNN implementations are based on streaming or a single computation engine, along with the compression and optimization strategies mentioned in Section 3.3. Compared with a single computation engine, the streaming architecture gains efficiency by pipelining the network and activating concurrent executions between layers. However, this efficiency leads to a resource burden and a long recompilation time because obtaining

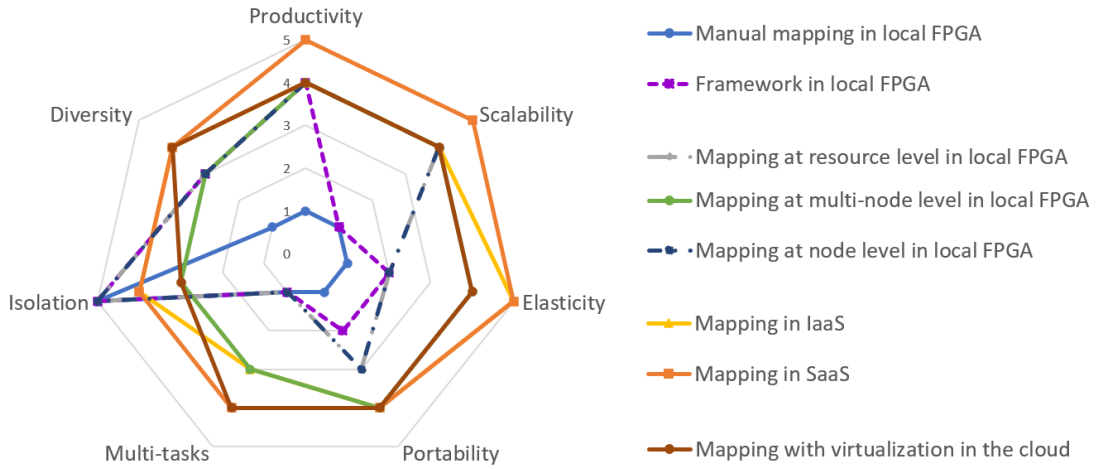


Figure 3.10 - Comparison of related methods with different characteristics.

a new CNN model requires regenerating the bitstream. According to the different requirements (e.g., resource constraints or speedup), researchers can choose different hardware structures in both the local and cloud FPGAs.

3.6 DISCUSSION

In the history of deploying CNNs on FPGAs, new requirements have been proposed at different stages, which has led to different challenges. With the development of a novel generation of platforms, technologies, and concepts, challenges have been resolved.

3.6.1 Unresolved challenges

Some challenges of using FPGAs in the cloud have not been fully resolved owing to their complexity. Here, we describe two major challenges: isolation and diversity.

3.6.1.1 Isolation

With the increasing efforts to provide a cloud environment for multiple tenants to deploy CNNs on the shared FPGAs, resources and performance isolation have become a concern in the cloud.

CNN accelerators on the FPGA usually run under full hardware access and may share resources. Therefore, malicious code can attack the entire platform for other tenants [45, 162]. Additionally, dataset collection can be time-consuming and expensive—particularly in industrial cases where datasets are of significant commercial value. Providing strict

data and resource isolation for multiple tenants can prevent unauthorised access to the dataset and avoid data leakage [160, 119].

Additionally, a CNN application may affect the performance of other CNN applications during concurrent execution [45, 64], which causes unreliable performance. However, few works [169, 168] discuss performance isolation problems, and their isolation remains underexplored.

3.6.1.2 Diversity

Diversity of CNN functions: Owing to resource limitations and development difficulties, the networks reported in the literature are standard (such as AlexNet and VGG) with common functions (such as convolution and pooling). With the continuous emergence of CNNs, the current CNN functions that can be implemented on FPGAs lack consistency with the development of CNN algorithms. However, the cloud environment provides more possibilities for exploring the deployment of CNNs with a rich set of functions on FPGAs by providing more resources and abstraction layers and can promote the diversity of CNN IP development.

Diversity of CNN usage: Training is a difficult phase to be performed on the FPGA, because all the features must be stored in memory until the corresponding errors are backpropagated, which requires more storage than inference. Existing works mainly focus on performing CNN inferences with relatively simple functions on the FPGA. Benefiting from the “unlimited” capacity and resources provided by the FPGA cloud, CNN training, fine-tuning, transfer learning, and the support of new functions in CNNs will be more feasible.

3.6.2 Industrial solution

To keep pace with the development of CNN accelerator design, novel platforms have been used in industry to enhance the hardware computing power. In 2019, Xilinx proposed a new SoC family called Versal, which is based on an adaptive compute acceleration platform, for accelerating applications such as CNNs. Versal tightly integrates software-programmable accelerators through the NoC structure, making accelerators scalable with flexible connections and achieving a high level of software abstraction for the rapid development of accelerators.

Xilinx also proposed a novel framework called Vitis AI [8]. The framework can be

interfaced with Caffe and TensorFlow and provides a unified solution, e.g., quantization, optimization, and pruning. Moreover, it allows the deployment of CNNs based on the ISA and can compile the latest CNNs into deep-learning processor unit instruction codes. Vitis AI can enhance the productivity and portability of CNN deployment, allowing software engineers to deploy CNNs without hardware expertise.

3.6.3 Roadblocks of FPGA Cloud

Solutions of FPGA-based accelerators in the cloud have been proposed for several years [2, 1]. Nevertheless, FPGAs have achieved less success compared to GPU and TPU architectures in the cloud. Deploying FPGA devices as easy-to-use resources in the cloud faces the following major roadblocks.

First, FPGA programming requires cloud users to have extensive hardware skills and expertise to deploy their applications in the cloud, which is a considerable challenge for software engineers and data scientists. Cloud providers must provide well-developed virtualization techniques for abstracting FPGAs [69]. As discussed in Section 3.5, virtualizing FPGAs in the cloud for artificial-intelligence applications still has issues, such as runtime overhead, multi-user support, user isolation, and data privacy. Additionally, the FPGA cloud provides users with high permissions to access the resources, where users can upload their bitstreams for application deployment, leading to malicious attacks and security problems [98]. Such problems hinder the success of FPGAs in cloud computing.

3.7 CONCLUSION

This paper summarizes several techniques to promote CNN deployments on FPGAs, including architectural design and optimization strategies. We reviewed related works based on FPGA virtualization and cloud deployment. Our study involved an in-depth analysis of the evolution of CNN deployment on FPGAs, from local FPGAs to virtualized FPGAs in the cloud. This topic was ignored by previous surveys.

With the rising concern regarding the adoption of FPGAs at the edge and in the cloud, porting CNNs onto FPGAs in cloud services will continue to attract attention in the years to come.

ACCELERATING CNNs ON FPGA PLATFORM

Contents

4.1	PRINCIPLE OF THE PLATFORM	51
4.2	STREAMING	53
4.2.1	<i>IP Design</i>	53
4.2.2	<i>Mathematical model of resource utilization</i>	59
4.2.3	<i>Latency results</i>	66
4.3	SINGLE ENGINE COMPUTATION	68
4.3.1	<i>IP design</i>	68
4.3.2	<i>Generic CNN parameters</i>	71
4.3.3	<i>Optimization</i>	71
4.4	QUANTIZATION	73
4.4.1	<i>Our quantization tool</i>	75
4.4.2	<i>Experimental results</i>	79
4.5	CONCLUSION	82

The objective is to propose a platform that can overcome IP design and validation challenges. As a result, machine learning engineers can infer any CNN model with their database without hardware expertise.

The platform has two main parts: The test vehicle and CNN IP.

The test vehicle is a specific module named by STMicroelectronics[®], accommodating various CNN IPs on FPGA devices. The responsibility of the test vehicle is to exchange data between external memory and a user-defined IP (in our context, CNN IP). The test vehicle is a fixed module, but it can be applied to different CNN IPs. Therefore, it is possible to partially synthesize the test vehicle and then lay it out in a fixed area in the placement & routing steps for the further implementation.

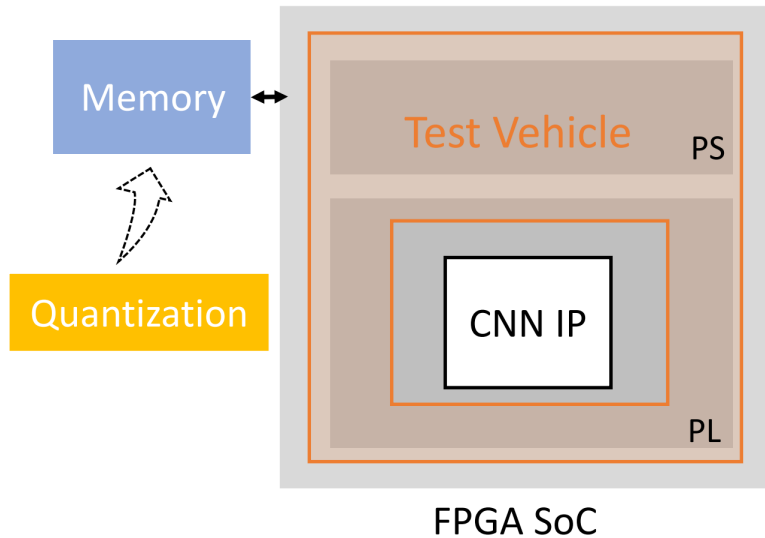


Figure 4.1 - Architecture of our platform.

The CNN IP is a general module referring to realize a part of CNN (single-engine IP) or the entire CNN (streaming IP). This IP can be seen as a black box and is synthesized independently of the platform. The CNN IP is designed to meet the different requirements and metrics of machine learning engineers. The machine learning engineers' requirements may be:

- A rapid execution on the CNN inference;
- Resource utilization as few as possible;
- Or, a trade off between execution time and resource usages.

Facing such a difference between requirements, the CNN IP in the platform thus

involves two types of hardware structures: streaming and single-engine computation. The streaming structure gains high throughput with a pipeline structure. In contrast, the single-engine structure is more friendly to large-scale CNN and consumes fewer resources than the streaming architecture.

The conception flow of the platform is shown in the figure 4.2. The test vehicle is synthesized to obtain the RTL schematic and resource utilization on the FPGA device, while the remaining available FPGA resources are reserved for CNN IP. At the same time, the parameters of the IP are selected (details of the parameters in the section 4.2.1.2). According to the given parameters and related mathematical models, the resource utilization of the IP is estimated. In the process of resource verification, if the remaining resources are insufficient to meet the resource requirements of the streaming IP, the machine learning engineers should either determine a new set of parameters or change the FPGA device. Otherwise, the streaming IP can enter the CNN integration process. Finally, the test vehicle and CNN IP are connected to generate the platform before the implementation.

In this chapter, the proposed platform aims at performing the inference of a general-purpose CNN. However, the platform is also designed for usages in other scenarios, presented in detail in chapter 5.

4.1 PRINCIPLE OF THE PLATFORM

The proposed platform consists of an integrated processing system (PS) and programmable logic (PL) to achieve the flexible implementation of each CNN IP. The data transmission of the CNN includes the transfer between FPGA and external DDR and the internal data transmission inside FPGA. The AXI4 protocol is applied for the transfers between FPGA and external DDR. Meanwhile, several types of transfer have been developed for internal transfer inside FPGA.

As shown in the figure 4.3, the PS part controls operations performed by the PL side in the classic system-on-chip application. PS also manages external communications (with DDR external memory and the ethernet link). The PS part:

- Gets back the results of the inference when available;
- Sends results to the machine learning engineer.

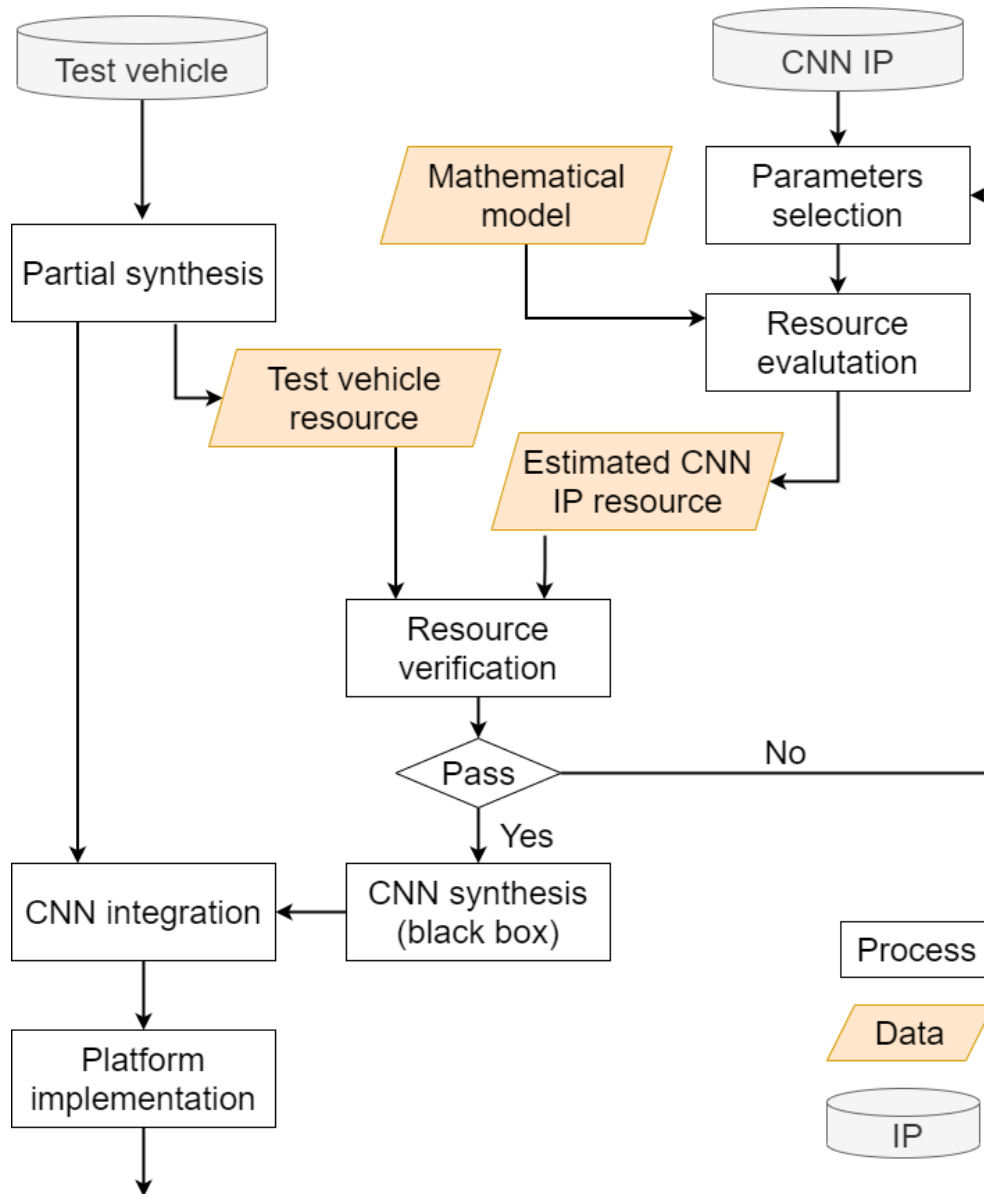


Figure 4.2 - Conception flow of our platform.

The control operations are done by a BareMetal application code executed on the ARM processor. This solution is faster than a full operating system.

The PL part is the CNN model itself. Using the SoC's PL side increases the system performance, reduces power, and delivers predictable latency for inference. As inputs and the output of any CNN model are unchanged, the CNN can be considered as a black box from the system point of view. It is connected to the machine learning engineer through the FPGA test vehicle.

Considering CNN as a black box, the synthesis tool turns this IP block off and creates a black box for inserting any CNN after the synthesis step. Thus, it enables configuring any

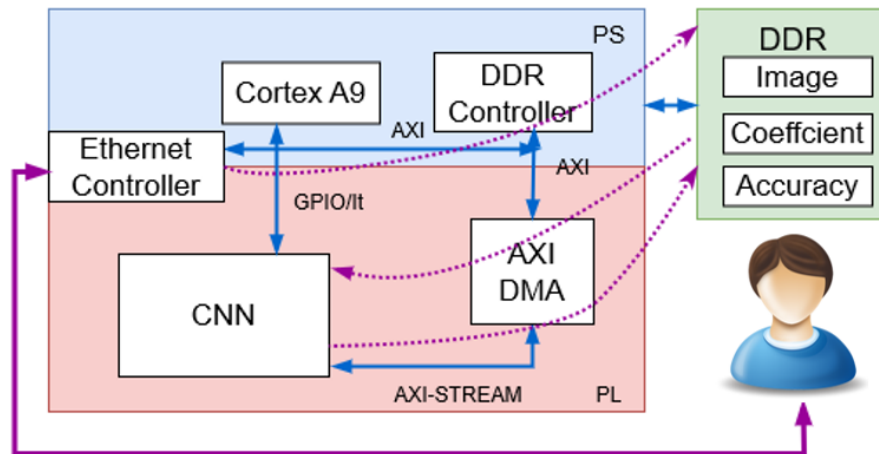


Figure 4.3 - The global structure of the FPGA-based platform.

CNN after implementing the FPGA test vehicle as long as the number of FPGA resources can handle the new model. The black box CNN has the following inputs and output:

- The input coefficients: weights and bias stored in the CNN before performing inference;
- The input data: corresponds to the pixels of the image being processed by the CNN (input for the inference);
- The output data: refers to the accuracy obtained at the output of the CNN (results of the inference).

4.2 STREAMING

The streaming IP usually consists of a complete CNN. The streaming IP implements a series of CNN functions, from the first convolutional layer to the last fully connected layer. This IP can generate a generic CNN architecture by defining the network parameters before generating the bitstream. The streaming IP should be modified to re-generate the appropriate CNN architectures if the CNN model changes.

4.2.1 IP Design

As shown in the figure 4.4, the streaming IP consists of several generic CNN operations, such as convolution, pooling, and fully connected blocks. The convolution block applies a line buffer to achieve efficient data transfer and computation. In addition, these blocks are interconnected with particular methods, which will be presented in the section 4.2.1.1.

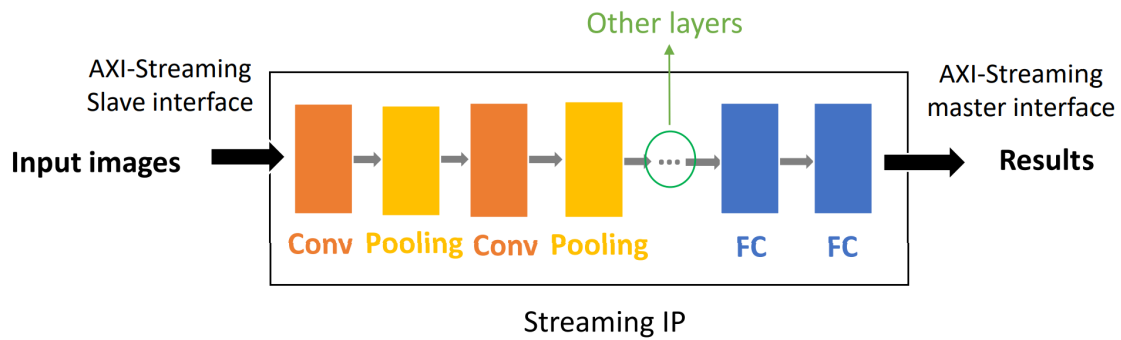


Figure 4.4 - Streaming CNN IP with AXI4-Streaming.

The streaming IP is interfaced by AXI4 protocol, consisting of two main interfaces to exchange data or results with the external memory:

- An AXI4 stream slave interface. It is designed for receiving data (input images) from the external memory;
- An AXI4 stream master interface. It is designed for sending data (processed results) to the external memory.

The final implementation of streaming-IP-based CNN is achieved in Xilinx[®] Vivado Design Suite Block Design Tool shown in figure 4.5. The module CNN_0 is the AXI4 interfaced streaming IP, which processes the computation of the whole neural network. The module processing_system7_0 is the processor of the FPGA SoC, which controls the data flow and configure the module AXI Direct Memory Access (DMA) and the module CNN_0.

Here, we added the AXI DMA module in the design to provides high-bandwidth direct memory access between the AXI4 Memory-mapped and AXI4-interfaced CNN IP without the processor[151]. The AXI4 DMA connects the input of the CNN IP by the Memory-mapped to Stream (MM2S) and the output of the CNN IP by the Stream to memory-mapped (S2MM), taking full advantage of the burst transmission mode. In addition, AXI DMA connects the processor in the PS side by S_AXI_Lite port for initialization or registers configuration.

4.2.1.1 Data transfer flow inside the streaming IP

Three communication structures are proposed to transmits pixels and coefficients inside the CNN architecture to efficiently accelerate the CNN operation. The line buffer structure is designed to calculate the convolution kernel, while the point-to-point and serial links are intended for the transmission of the data in the entire network.

- **Line buffer:** The sliding window with line buffer shown in the figure 4.6 contains a window $K \times K$, where K is the convolution size. This sliding window uses a chain of shift registers to store the previous pixels. The later come pixel is transmitted to the first register and pushes the previous pixel forward in the line buffer. Given the image size of $I \times I$, the minimum required registers are $I \times (K-1)$. The $(I-1) \times (K-1)$ clock cycle is required to output the first valid convolution result from the first pixel into the line buffer. From this moment on, the line buffer will generate a new convolution result per clock cycle. We also apply a pixel counter to identify the valid output results among all outputs in this structure. Such a structure effectively reuses the same convolution kernel and stores part of the feature map in the buffer, greatly reducing the external memory's access requirement.

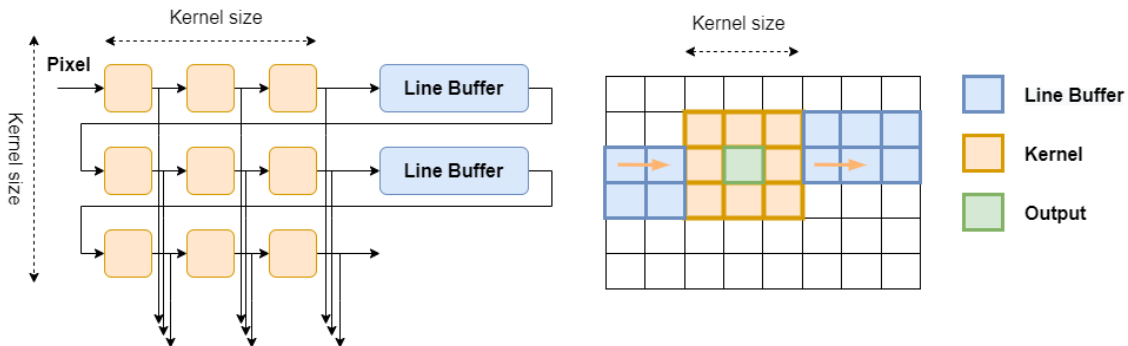


Figure 4.6 - Data flow of extract neighborhood pixels in with line buffer.

- **Point-to-point communication:** The data structure is a point-to-point connection between blocks shown in the figure 4.7. This structure fully unrolls the succession of the layers and applies a straightforward implementation. This structure allows streamlining communications between each neuron layer by layer. Such a structure with pipeline mode can make full exploration of layer parallelism of CNN structure and accelerate execution time of CNN. However, the layers are unfolded in full parallel, which causes insufficient hardware resources problems.

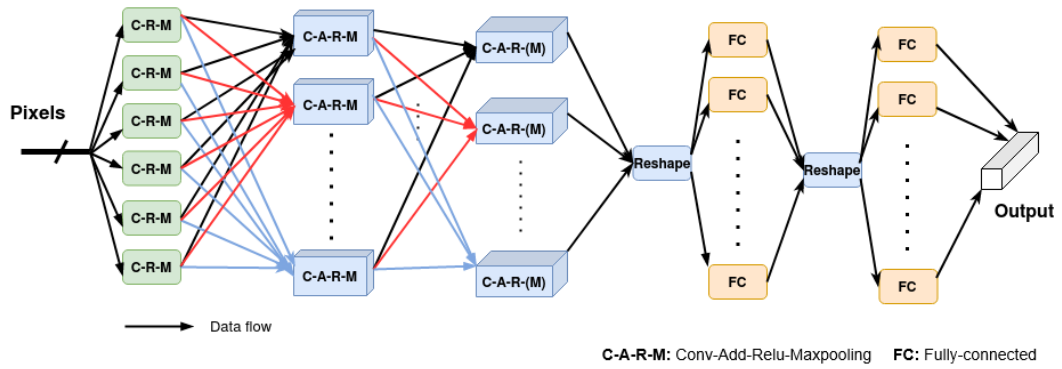


Figure 4.7 - Data flow of loading pixels in point-to-point mode.

- **Serial link:** The structure for loading coefficients inside the CNN is a serial link that goes through all Convolution blocks, from the 1st layer to the last layer shown in the figure 4.8. The input coefficient of the next block is connected to the output coefficient of the previous block, and its function is similar to first-in-first-out. Therefore, the coefficients of the fully connected layer should be loaded into the block first to ensure the correctness of the calculation. Such a structure brings an overhead but does not require direct external access to the system address and data buses. Thus, FPGA resources and interconnect are minimized to be fully used by the IPs and the data flow structure.

The coefficients which are sent to the blocks by a serial link are detailed in the figure 4.9: The first coefficients are those of the last layer, weights, and bias, must be sent first until the last ones (coefficients of the first layer). A tool is also developed in this part to order these coefficients extracted from the Tensorflow framework.

4.2.1.2 Generic network parameters

The streaming architecture can be configured through the following generic parameters. These parameters aim at generating flexible CNN structures from the fixed template.

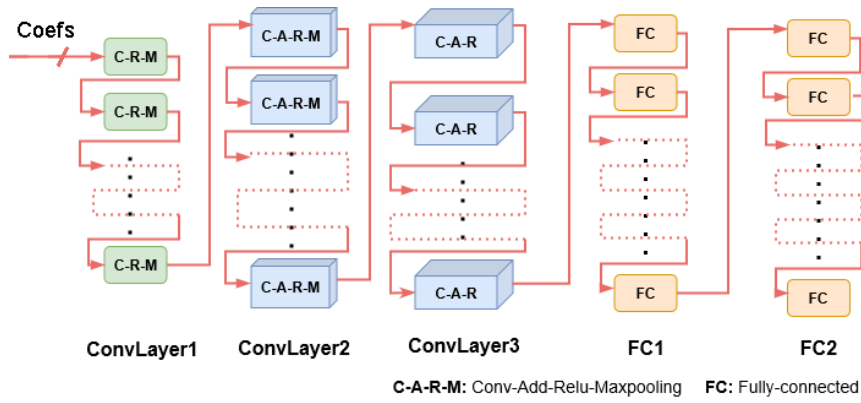


Figure 4.8 - Data flow of loading coefficients in serial link.

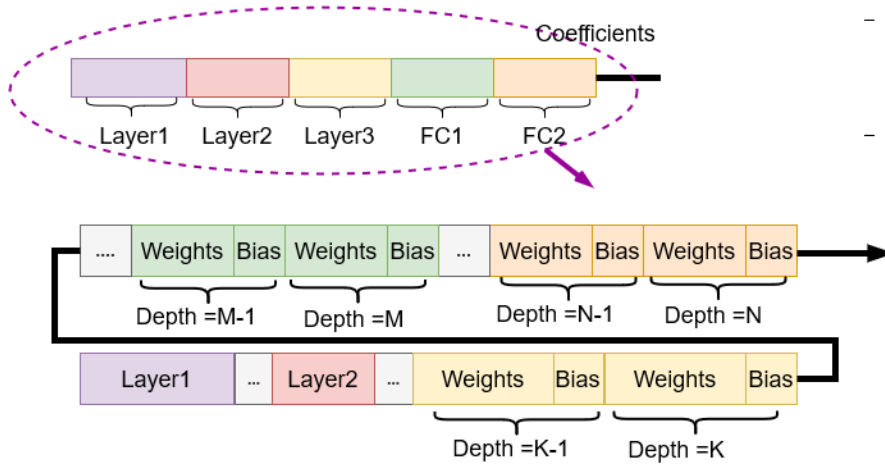


Figure 4.9 - Principle of weights and bias ordering.

The parameters are defined in the VHDL package, which can only be updated before generating the bitstream.

- **D**: Data width (number of the bit) of the image. This parameter can be divided into the integer part **D_{int}** and the decimal part **D_{dec}**;
- **C**: Data width (number of the bit) of the weights and bias. This parameter can be divided into the integer part **C_{int}** and the decimal part **C_{dec}**;
- **L1, L2 and L3**: Channel size of the first, second, and third convolution layers;
- **F1, F2**: Channel size of the first and second fully-connected layers;
- **K*K**: Convolution filter size. This parameter can be configured to any size;
- **P*P**: Pooling operation size. This parameter can be configured to any size;
- **I*I**: Input image size. This parameter can be configured to any size.

4.2.2 Mathematical model of resource utilization

This section aims to predict the IP resource utilization for a given set of network parameters. In more detail, we first select specific network parameters to obtain the synthesis results of streaming IP (especially the resource utilization results). Then, several mathematical models are extracted, which describe the relationship between parameters and resources utilization. As a result, we reach the goal of predicting resource usage when entering any network parameters.

The synthesis time of the CNN IP by a synthesis tool (e.g., Xilinx Vivado synthesis tool) takes long times according to different configurations. For example, it takes one hour to synthesize a standard LeNet5 with 16-bit data width in Vivado 2018.3. Therefore, it is not practical to launch many synthesis once the configuration changes. However, the prediction of the resources utilization can help find a suitable configuration in different use cases proposed in chapter 5 without going through the synthesis process.

The process can be described as shown in figure 4.10:

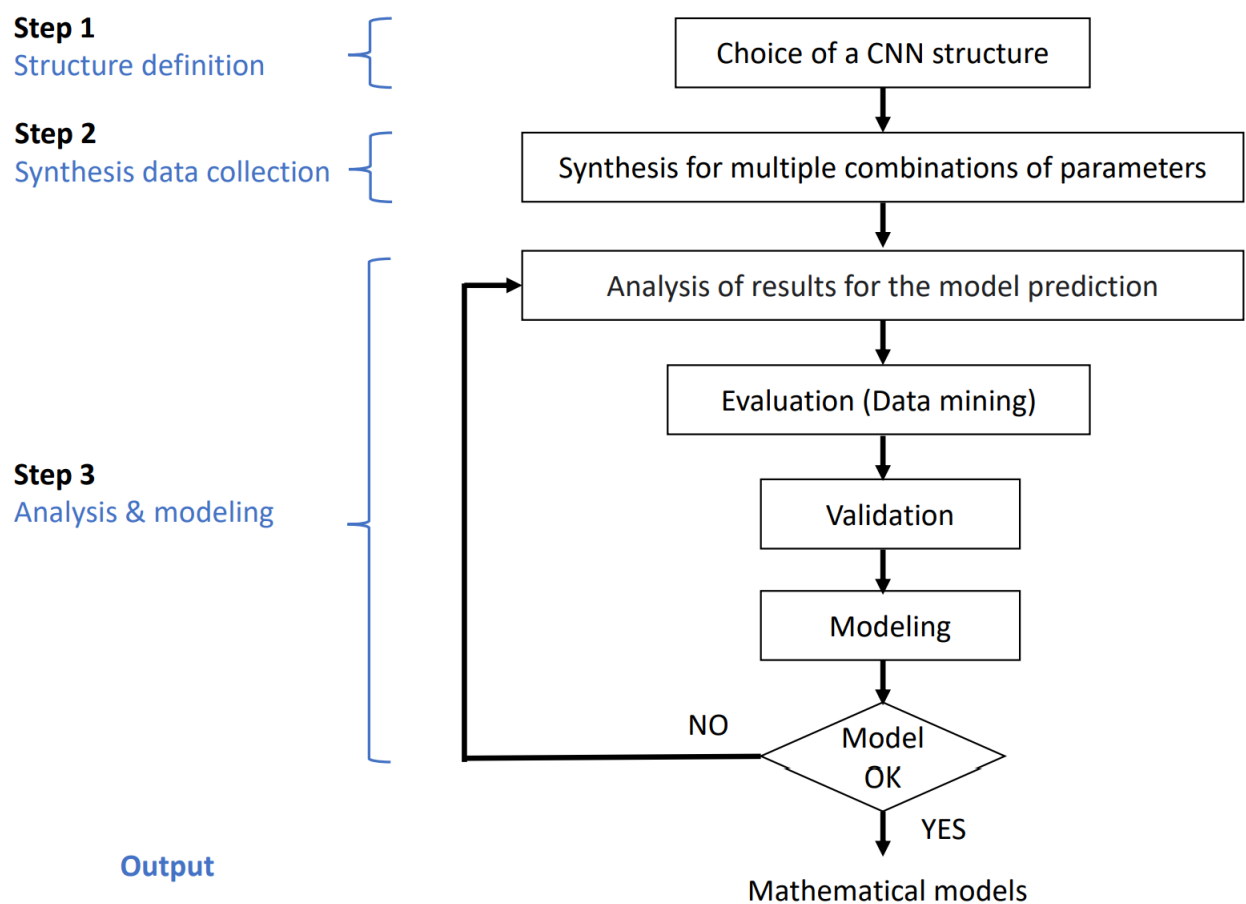


Figure 4.10 - Method for constructing the mathematical models

- **Step 1** Structure definition: In this step, we first choose a network structure as the input of the streaming IP (e.g., data width, layer number, network channel). The streaming IP is written in VHDL, and the streaming IP can configure all CNNs with standard functions. Then, we determine the varying parameters of the network.
- **Step 2** Synthesis data collection: In this step, the data refers to the resource utilization regarding different network parameters provided in step 1. The data is extracted from the synthesis reports produced by Xilinx Vivado 2018.3. The data may change due to different tool versions or FPGA devices, but always have the same types: Lut, Block ram, DSP, and Flipflop. To facilitate and automatize the synthesis process, we used a script developed in the laboratory. This script can continuously take a set of network configurations as IP's input and automatically launch the synthesis process in Xilinx Vivado. Finally, the script can extract all synthesis reports and re-organize the synthesis results in excel for the following step data analysis.
- **Step 3** Synthesis data analysis and modeling: In this step, we will conduct the relation between resources utilization and network parameters and examine whether the mathematical models can be extracted and integrated into the platform.

First, the correlations of these data calculated by Rstudio software are evaluated. Then, if the correlation exists between two data, we will find the network parameters that have a significant impact on the use of resources and extract the mathematical model. Finally, given the same network parameters, we compare the resources predicted by the mathematical model with the synthesized results to verify the correctness of the extracted mathematical model.

The experiments use about 800 configurations to evaluate the resources. I, K, and P are considered constants.

4.2.2.1 Resource utilization based on data width

As quantization is commonly used to optimize FPGA resources and speed up the inference time, the first evaluation is based on varying the data width on a CNN architecture.

The experiment of varying the data width (data width represents pixel size + weight size) is conducted from 8 bit to 32 bits, with L1, L2, L3, FC1, and FC2 being the constant. Figure 4.11 shows the evaluation of the resources of DSPs and logic LUT. We can observe that:

- The use of LUT as logic and DSP is mutually constrained. For the same network generated by our framework, DSP use will significantly reduce the use of LUT and inversely.
- Data width has a significant impact on the use of DSPs due to the DSP hardwired in FPGAs. In our experiment, the synthesis tools will implement DSP blocks with data higher than 16-bits. Therefore, below this value, the tools only use Logic LUTs instead.
- With the default synthesis settings, the threshold of using DSP depends on the data width in hardwired DSP blocks integrated into the FPGA. Taking Xilinx Virtex7 VC707 as an example, data width over 22-bit will use DSP to perform convolution operations, while data width less than 22-bits use only LUTs in our framework. While using Xilinx Zynq UltraScale as the platform, our framework deploys the convolution in logic LUTs when the data width is less than 16-bits.

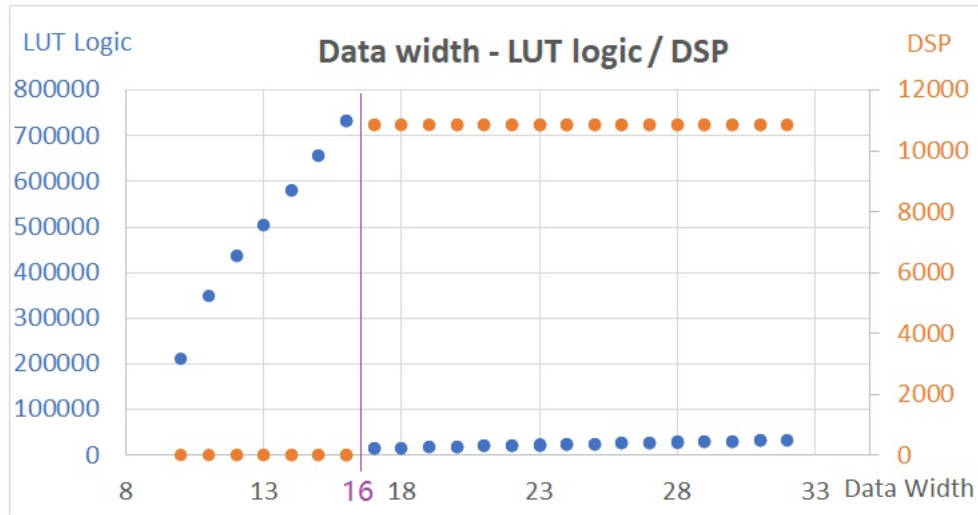


Figure 4.11 - Effect of data width on Logic LUT and DSP.

4.2.2.2 Resource utilization based on other network parameters

In this part, we study the impact of CNN parameters (L1, L2, L3, FC1, and FC2) change on FPGA resource usage. The mathematical models are extracted based on these parameters. Since [41] proves that CNN with a small data width provides high accuracy, the pixel and weight widths in this section are all below 16 bits. DSP is ignored in this section because the default setting of the synthesis tool is not to use DSP to realize the implementation.

The primary goal of the L1-L2-L3 layer is to detect the local features of the previous layer, FC1-FC2 is used for label prediction. Therefore, the resource utilization analysis of neural networks is divided into two parts accordingly.

The first part deals with the correlation between Layer1/Layer2/Layer3 and FPGA resources. The data width is constant (Pixel width = 4bits, weight width=4bits), while layers L1, L2, and L3 vary. The objective is to mathematically model the relation between the CNN input parameters and material resources used without going into the synthesis process.

Table 4.1 - Pearson's correlation (>0.7) extracted from R.

	Logic LUT	Memory LUT	Flipflop
Layer1	–	0.88	–
Layer2	0.75	–	0.76
Layer3	0.75	–	0.75

Table 4.1 presents Pearson's correlation extracted from R. When the Pearson's correlation is higher than 0.7, the closer the points are located to another one on the line. Thus, a correlation higher than 0.7 indicates that it is possible to obtain some linear mathematical models to evaluate FPGA resources based on CNN parameters.

By analyzing similarities between groups in the table 4.1, we can conclude that the changing parameter of Layer1 has an essential effect on the utilization of Memory LUTs. In contrast, the utilization of Logic LUT and Flipflop mainly depends on Layer2 and Layer3. The impact is summarized as follows:

- $Number_{MemoryLUT} = f(Layer1)$,
- $Number_{LogicLUT} = f(Layer2, Layer3)$,
- $Number_{Flipflop} = f(Layer2, Layer3)$.

According to Pearson's correlations, we draw the figures of 4.12, 4.13, 4.14, 4.15 and 4.16 (X-X-X-X-X represents the value of Layer1-Layer2-Layer3-FC1-FC2).

The figure 4.12 first indicates that the utilization of Memory LUT increases linearly with the increase of Layer1 since Pearson's correlation is 0.88 between Memory LUT and Layer1. Afterward, it does not affect the Memory LUT utilization when Layer3, FC1, and FC2 are halved. Only changes in Layer2 will cause a proportional change in resource

utilization. The equation of the resource utilization depending on different Layer1 can be identified as follows:

$$Number_{MemoryLUT} = 275.45 * Layer1 + 64.326 \quad (Layer2 = 16) \quad (4.1)$$

$$Number_{MemoryLUT} = 147.22 * Layer1 + 32.663 \quad (Layer2 = 8) \quad (4.2)$$

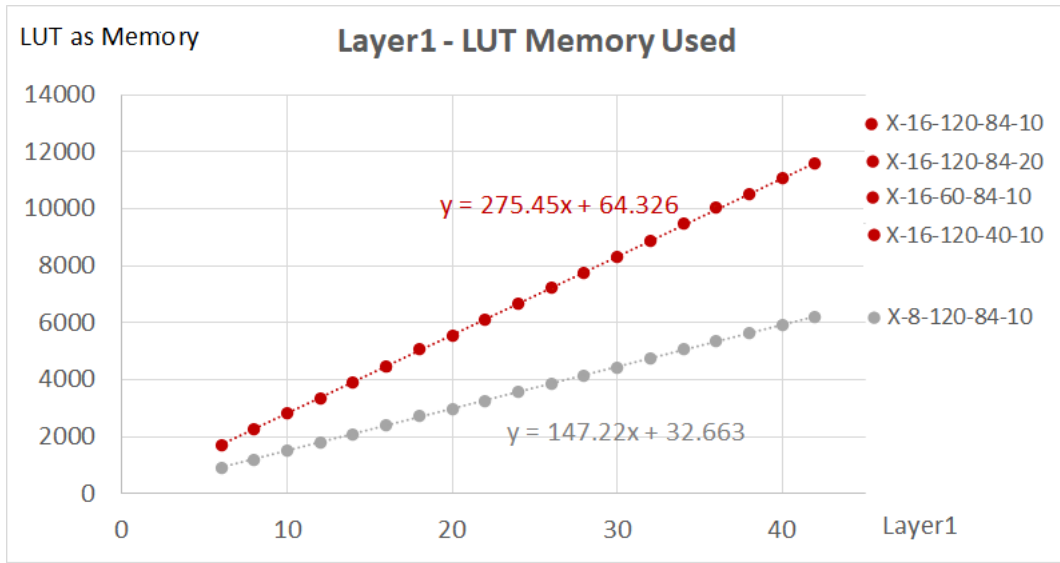


Figure 4.12 - Mathematic models of LUT Memory used in terms of Layer1

The figure 4.13 and 4.14 first show that the utilization of Logic LUT and flipflop increase linearly as the number of Layer2 increases. Next, we can observe that although Layer1 and F1 are halved, the utilization of Logic LUT and Flipflop remain unchanged. However, the utilization of Logic LUT and Flipflop are halved if Layer3 is halved. The equations can be summarized as follows:

$$\begin{aligned} Number_{LogicLUT} &= 36493 * Layer2 + 128934 \\ Number_{Flipflop} &= 27433 * Layer2 + 93230 \\ (Layer1 = 12, Layer3 = 120, FC1 = 84, FC2 = 10) \end{aligned} \quad (4.3)$$

$$\begin{aligned} Number_{LogicLUT} &= 34791 * Layer2 + 126229 \\ Number_{Flipflop} &= 26020 * Layer2 + 91130 \\ (Layer1 = 6, Layer3 = 120, FC1 = 84, FC2 = 10) \end{aligned} \quad (4.4)$$

$$\begin{aligned} Number_{LogicLUT} &= 34703 * Layer2 + 61604 \\ Number_{Flipflop} &= 26020 * Layer2 + 44964 \\ (Layer1 = 6, Layer3 = 120, FC1 = 40, FC2 = 10) \end{aligned} \quad (4.5)$$

$$\begin{aligned} Number_{LogicLUT} &= 18255 * Layer2 + 71728 \\ Number_{Flipflop} &= 13741 * Layer2 + 50411 \\ (Layer1 = 6, Layer3 = 120, FC1 = 84, FC2 = 10) \end{aligned} \quad (4.6)$$

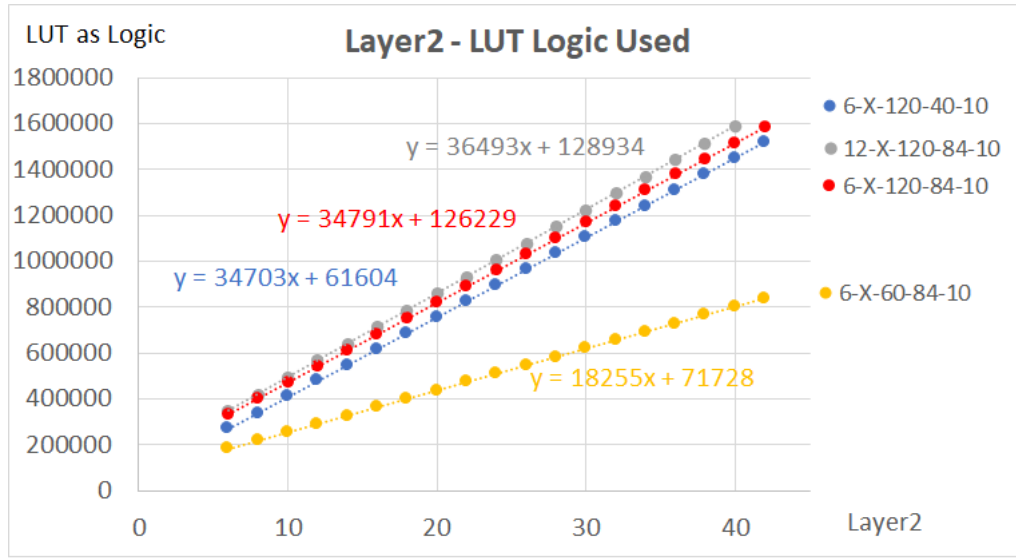


Figure 4.13 - mathematic models of LUT Logic Used in terms of Layer2

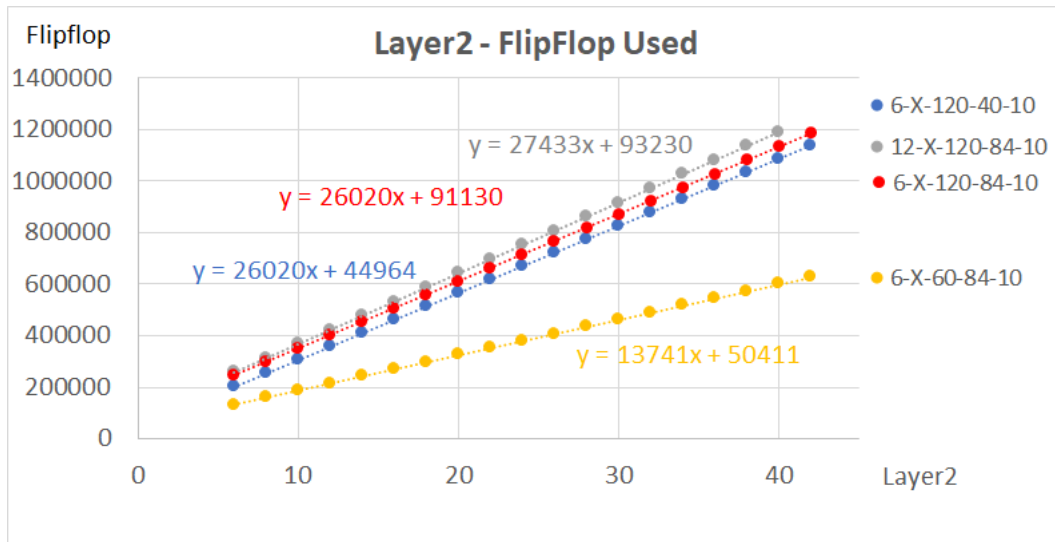


Figure 4.14 - Mathematic models of flipflop Used in terms of Layer2

The figure 4.15 and 4.16 show similar results. The resource utilization of Layer 3 increases proportionally. In addition, in the case of a given value of Layer3, Layer2 has the most critical impact on the utilization of logic LUTs and flip-flops. The equations can be summarized as follows:

$$\begin{aligned} \text{Number}_{\text{LogicLUT}} &= 5332.9 * \text{Layer3} + 46014 \\ \text{Number}_{\text{Flipflop}} &= 3944.7 * \text{Layer3} + 33684 \\ (\text{Layer1} = 6, \text{Layer2} = 16, \text{FC1} = 84, \text{FC2} = 10) \end{aligned} \quad (4.7)$$

$$\begin{aligned} \text{Number}_{\text{LogicLUT}} &= 4822.5 * \text{Layer3} + 40025 \\ \text{Number}_{\text{Flipflop}} &= 3591.8 * \text{Layer3} + 29880 \\ (\text{Layer1} = 6, \text{Layer2} = 16, \text{FC1} = 40, \text{FC2} = 10) \end{aligned} \quad (4.8)$$

$$\begin{aligned}
Number_{LogicLUT} &= 5309.9 * Layer3 + 79203 \\
Number_{Flipflop} &= 3944.4 * Layer3 + 58469 \\
(Layer1 = 12, Layer2 = 16, FC1 = 84, FC2 = 10)
\end{aligned}
\tag{4.9}$$

$$\begin{aligned}
Number_{LogicLUT} &= 3129.7 * Layer3 + 30745 \\
Number_{Flipflop} &= 2310.6 * Layer3 + 21560 \\
(Layer1 = 6, Layer2 = 8, FC1 = 84, FC2 = 10)
\end{aligned}
\tag{4.10}$$

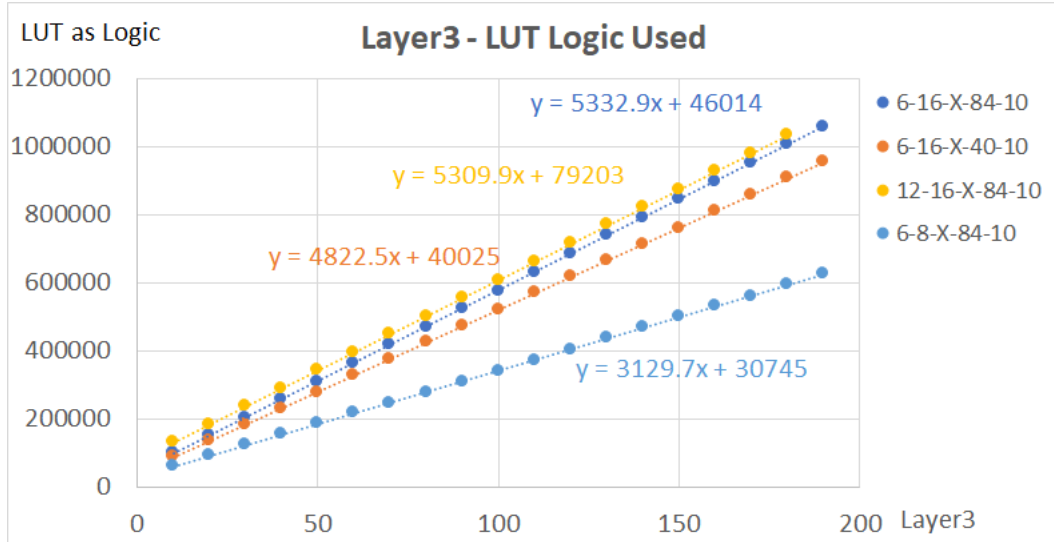


Figure 4.15 - Mathematic models of Logic LUT in terms of Layer3.

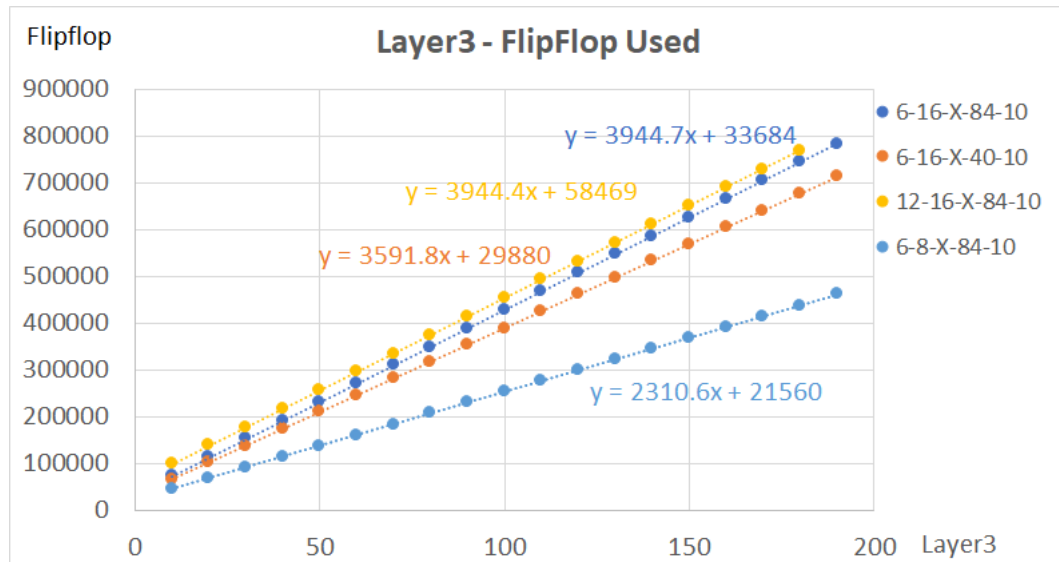


Figure 4.16 - Mathematic models of Flipflop in terms of Layer3.

The second part extracts the models between fully connected and FPGA resources. Again, Pearson's correlation is above 0.70, indicating a linear relationship between fully connected layers and FPGA resources. The impact is summarized as follows:

- $Number_{MemoryLUT} = constant$,

- $Number_{LogicLUT} = f(FC1, FC2),$
- $Number_{Flipflop} = f(FC1, FC2).$

The equations of the resources utilization are described as follows:

$$\begin{aligned} Number_{LogicLUT} &= 1487.8 * FC1 + 558089 \\ Number_{Flipflop} &= 1050.5 * FC1 + 418777 \\ (Layer1 = 6, Layer2 = 8, Layer3 = 120, FC2 = 10) \end{aligned} \quad (4.11)$$

$$\begin{aligned} Number_{LogicLUT} &= 1388.2 * FC2 + 669744 \\ Number_{Flipflop} &= 680.05 * FC2 + 500199 \\ (Layer1 = 6, Layer2 = 8, Layer3 = 120, FC1 = 84) \end{aligned} \quad (4.12)$$

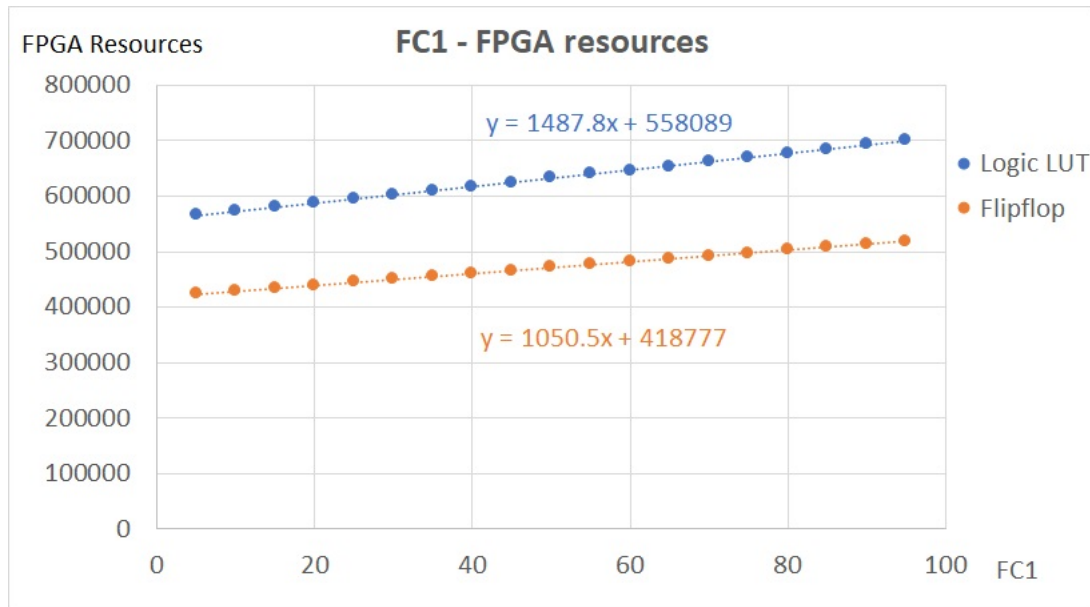


Figure 4.17 - Mathematic models of FPGA resources in terms of FC1.

The above mathematical models use Layer1/2/3 and FC1/2 as individual variables to explore the relationship with FPGA resources used. These models are accurate as the coefficient of determination that evaluates the regression model is $R=1$. Thus, machine learning engineers can use the mathematical model to quickly determine or adjust CNN parameters under the resources of available FPGAs.

4.2.3 Latency results

Each CNN model requires N clock cycles to load N coefficients. N depends on CNN configuration. Based on the three configurations in Table4.2, Table4.3 lists the timing evaluation of loading coefficients and performing inference in each CNN model. The working frequency is 150 MHz.

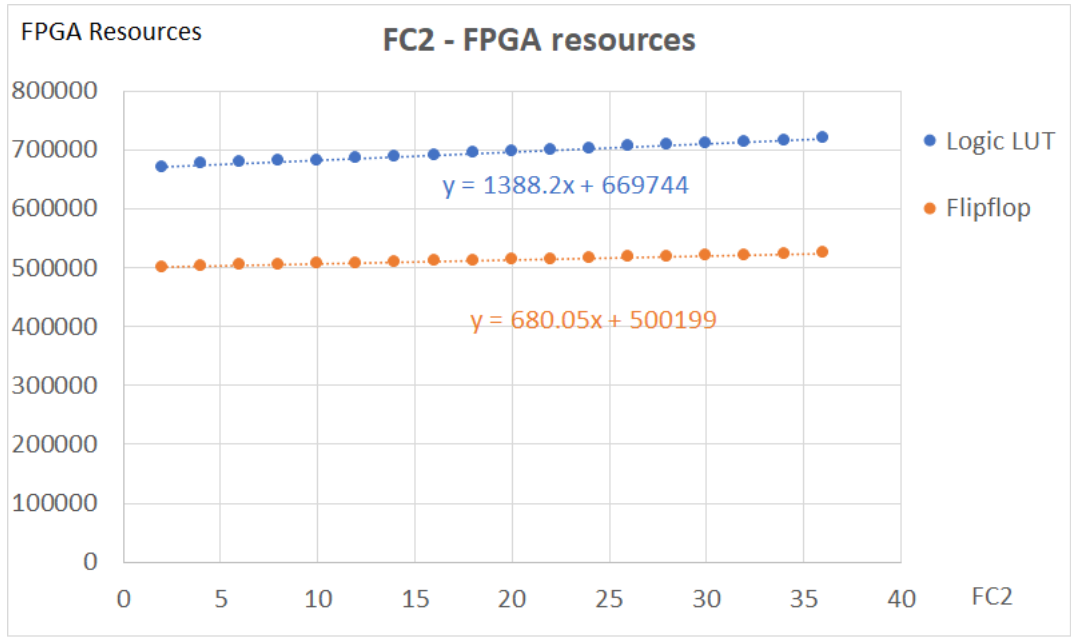


Figure 4.18 - mathematic models of FPGA resourceS in terms of FC2

Table 4.2 - Configuration of each CNN

Config	D_{int}	D_{dec}	C_{int}	C_{dec}	L1	L2	L3	F1	F2
1	4	2	2	2	6	16	120	84	10
2	4	2	2	2	6	16	60	40	20
3	4	2	2	2	12	8	60	84	10

Table 4.3 - Timing evaluation of loading coefficients and performing inference.

Config	Loading Coefficients		Performing Inference	
	Clock cycles	time (μ s)	clock cycles	time(μ s)
1	13788	92	3175	21.1
2	7883	52	1190	7.9
3	8560	56	1291	8.5

These CNN models only take a few microseconds to perform inference on one image and load coefficients. The estimated times depend on the CNN structures. Generally, the test set for the inference includes 1000 to 100000 images, so it takes a few seconds to a few minutes to execute the inference for the dataset. In our estimation, the time to load images to DDR is not considered depending on the communication protocol. Nevertheless, pipeline technology can efficiently accelerate the global inference process.

4.3 SINGLE ENGINE COMPUTATION

Due to the large number of layers and computational complexity of the standard CNN, it is difficult to map each layer of the entire CNN structure inside the FPGA. Therefore, the current mainstream method is to adopt the acceleration layer by layer, namely single-engine computation IP. This method sequentially executes the layers of the entire CNN on the FPGA, and restores the output data of the current layer to the external memory. When calculating the next layer, the output result of the previous layer will be read back to the single engine calculation for the calculation of the current layer.

4.3.1 IP design

As shown in the figure 4.19, the single engine computation IP consists of convolution and pooling operation. The convolution operation can be configured as the fully-connected operation by changing the software code inside the FPGA PS.

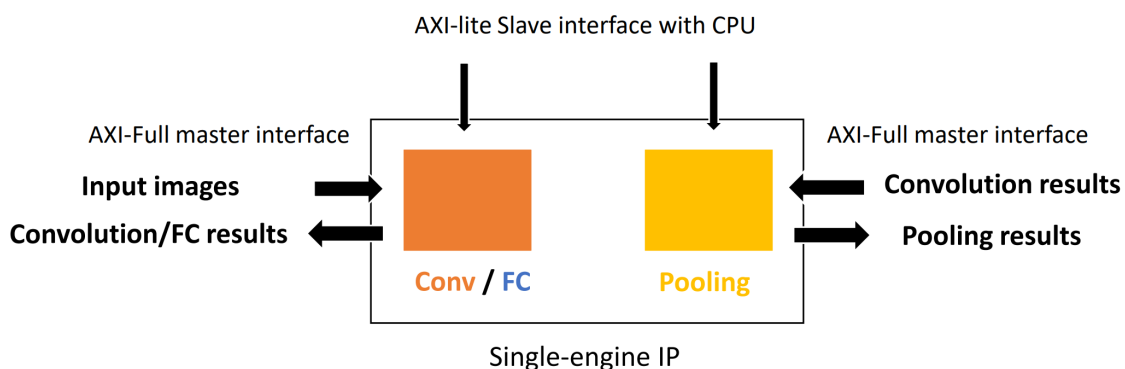


Figure 4.19 - Single engine CNN IP with AXI4-full standard.

The single-engine computation is interfaced by AXI4 protocol, basing on the burst transmission mode and uses a two-way handshake mechanism. When the valid and ready signals are high at the same time, the data transmission starts. Data bandwidth between PL and PS is 32-bit for the single-engine IP. It consists of two principal interfaces to exchange data or results with the external memory:

- An AXI4-full master interface. It is designed for receiving data (input feature images) from the external memory to convolution or pooling operation, and sending data (processed results) to the external memory;
- An AXI4-lite slave interface. It is designed for configure the convolution and pooling operations (e.g., kernel size, padding or not padding).

The final implementation of single-engine IP-based CNN is achieved in Xilinx[®] Vivado Design Suite Block Design Tool shown in the figure 4.20. The module pooling_0 is the AXI4-interfaced IP, which is responsible for the pooling operation in each layer. The module conv_0 is the AXI4-interfaced IP, aiming at process the convolution and fully-connected operations in the each layer. These two IPs are written in C language in the Xilinx[®] Vivado HLS. The module processing_system7_0 is the processor of the FPGA SoC, which controls the data flow and configure the module pooling_0 and conv_0 with different network parameters. The processor is also responsible for configuring the module conv_0 to operate convolution or full connection.

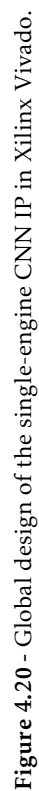


Figure 4.20 - Global design of the single-engine CNN IP in Xilinx Vivado.

4.3.2 Generic CNN parameters

The single-engine architecture can be configured through the following general parameters. These parameters are designed to generate flexible CNN functions (convolution, pooling, and full connection) from a fixed template. The parameters of the function are determined in the software code, even after the code stream is generated, the parameters can be updated multiple times in the software code.

- **D**: Data width (number of the bit) of the image. This parameter can be divided into the integer part **Dint** and the decimal part **Ddec**;
- **C**: Data width (number of the bit) of the weights and bias. This parameter can be divided into the integer part **Cint** and the decimal part **Cdec**;
- **C_in**: Number of input feature map channels;
- **C_out**: Number of output feature map channels;
- **S** : Stride of the convolution. This parameter can be configured to any size;
- **Padding**: Padding or not. This parameter can be configured to 0 (without padding) or 1 (with padding);
- **K*K** : Convolution kernel size. This parameter can be configured to any size;
- **P*P** : Pooling operation size. This parameter can be configured to any size;
- **I*I**: Input image size. This parameter can be configured to any size.

4.3.3 Optimization

Vivado HLS provides the annotation of the C code with the `#pragma` directives to obtain several optimizations or implementation. The directives can be classed into:

- Interface (e.g., function-level interface, port-level interface, AXI4 interface) to specify how RTL ports are created from the function definition during interface synthesis [153];
- Data and control flow (e.g., loop unrolling, dependence, pipeline, inlining, instantiation) to change the data flow to improve resource utilization or execution time;

- Storage allocation (e.g., memory type) to choose the correct inferred circuit (Block RAM, Distributed RAM, Shift Register).

The data and control flow directives are usually placed in a loop or if-else branch to change the synthesis results of the algorithm. From the pseudo code of the convolution operation, we can observe three different levels of loops. To optimize these loops, we apply the method pipeline and unroll based on [172]. The detail of these two directives can be concluded as follows:

- Pragma pipeline, which enables concurrent execution of the loop, thereby reducing overall latency of the loop. Moreover, it provides the initiation interval (II) between the pipelined in case of facing dependencies or hardware resource constraints issues. We apply the pipeline strategy inside each convolution kernel to pipeline the reading pixels process. As a result, the pixel can be processed each clock cycle.
- Pragma unroll, which unrolls the loop and creates independent operations, thereby increasing the throughput of operations. In addition, it provides an expansion factor to perform functions in multiple iterations. However, unroll can only be applied to tasks that have no dependencies between loop iterations.

We use a factor to apply the expansion strategy to the input and output channels. This strategy obtains parallel computations of a set of input images, thereby avoiding spending a lot of time processing each image sequentially.

Besides, we apply Pragma array partition to the input and output arrays. By default, the input and output arrays will be mapped to BRAM by Vivado HLS. However, BRAM can only be expanded to two ports for reading and writing, limiting the throughput. Using directive array partition can split the input and output into sub-arrays and then execute them simultaneously, thereby increasing the parallelism of the convolution.

The final optimization can be explained the listing 4.1:

Listing 4.1 - Convolution operation with different directives.

```
#Pragma HLS array_partition variable=din complete dim=1
#Pragma HLS array_partition variable=dout complete dim=1

for (co=0; co<C_out; co=co+1)    //loop of output channel C_out
#Pragma HLS unroll

    for (h=0; h<H_out; h=h+1)      //loop of the output heigh H_out
```

```

for (w=0; w<W_out; w=w+1)          //loop of the output width W_out
{
    new sum=0;

    for (ci=0; ci<C_in; ci=ci+1)      //loop of the input channel
    #Pragma HLS unroll

        for (r=0; r<K; r=r+1)        //loop of kernel heigh K

            for (s=0; s<K; s=s+1)      //loop of kernel width K
            #Pragma HLS pipeline II=1

                sum+=din[h*S-P+r][w*S-P+s][c]*wt[r][s][c][co];

    dout[h][w][k] =sum;
}

```

The synthesis result is carried out by Vivado HLS 2018.3 on Xilinx zynq 702. The working frequency is about 100Mhz. The table 4.4 shows the latency results in terms of different directives, where $C_{in}=10$, $C_{out}=8$, $W_{out}=H_{out}=6$, $K=5$.

Table 4.4 - Comparison of latency results (clock cycle) under different directives.

Directive	Default	Unroll	Pipeline & Unroll
Latency	995441	275221	52831
Pipeline II	-	-	5

4.4 QUANTIZATION

Several methods are proposed to compress the network size based on the sharing of weights, network pruning, network quantification, etc. Quantization is a powerful solution that efficiently reduces memory bandwidth, power consumption, and computation time. Related works on quantization are divided into two categories in the figure 4.21:

1. Post-training quantization in which a pre-trained floating-point model on GPU/CPU is converted to a fixed-point model;
2. Quantization with aware-of- training in which a pre-trained floating-point model on GPU/CPU is converted to a fixed-point model with fine-tuning on GPU.

Post-training quantization is simple and easy to realize in real-world applications and can quickly quantify the network. In the absence of a dataset, the works applying post-training quantization typically aim at minimizing some surrogate errors introduced during the quantization process (e.g., round-off errors) to the end-to-end loss. In 2016,

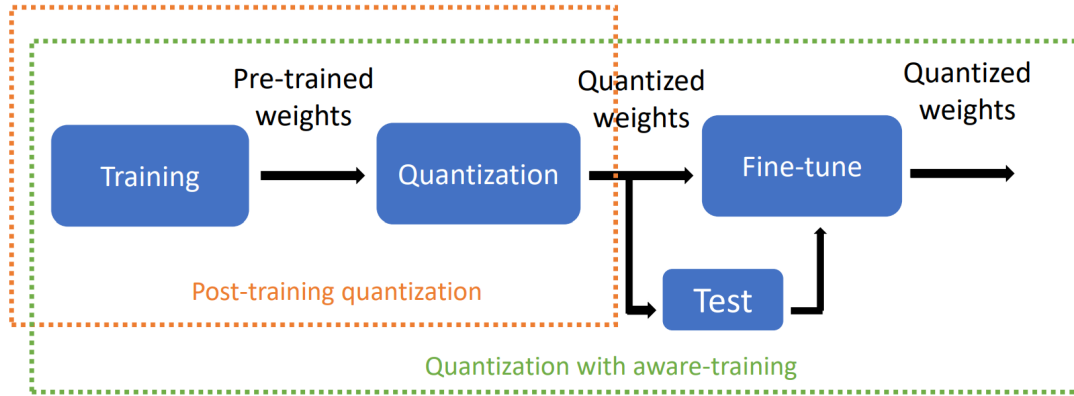


Figure 4.21 - Different quantization methods.

Lin D et al.[88] apply an optimal 16-bit quantization on cifar-10 based on signal-to-quantization-noise-ratio (SQNR). The experiments prove that the naïve method of quantizing networks (e.g., Uniform quantization) results in a subpar performance in error rates relative to the SQNR method. In 2017, a new SDK proposed by Nvidia, named TensorRT[100], has proved the feasibility of an 8-bit quantization of forward pass. Migacz proposes to utilize Kullback-Leibler(KL) divergence in determining the linear range to apply layer quantization. As quantizing weights and biases under 8-bit precision without retraining introduce a significant loss in accuracy, Choukroun. Y et al[29] calculate the clipping values using Minimum mean squared error (MMSE) in 2019. The method is kernel-wise for weights and channel-wise for activation. First, they address 4-bit linear quantization problems for the state-of-art networks (e.g., Alexnet, Resnet50, inception V3, DenseNet). In 2019, Ron. B et al. [20] introduce an analytical clipping for the integer quantization approach, which reduced the local error introduced during the quantization process. Combined with layer-wise bit allocation proposed by Sajid, they successfully compress VGG, inception v3, res18 into 4-bit precision with just a few percent less than baseline inaccuracy. More works have a focus on optimizing the quantization performance without training. In 2018, Kim.D et al.[76] proposed generalized gamma distribution (GGD) to obtain the optimal quantization steps. In 2019, Meller.E et al.[99] exploit the scale equivariance of network function to rescale weights channel and quantize ResNet-50, inception V3 by weight factorization.

Quantization with fine-tuning is a powerful approach to compensate for quantization-induced errors using a complete dataset, extremely low-bit quantization. In 2015, Sajid Anwar et al.[17] proposed a layer-wise sensitivity analysis for non-uniform weights

quantization for Most and Cifar-10. L2 error minimization is applied to find the optimum quantization level for each layer. In every iteration, they quantized weights of only one layer to low precision and calculated the network output. Other layers are kept in high accuracy to compensate for the quantization error. The computed change in quantized weights is added to high precision weights. In 2016, a fast and automated framework improved quantification method was proposed by Philipp Gysel et al.[55]. They quantized not only layer weights but also activations for CaffeNet and SqueezeNet with a maximum error tolerance of 1 %. Later on, other works have been invested in training the network with lower precision, including but not limited to BinaryNet[31], Xnor-Net[115],DOREFA-Net[182], which achieve promising results. BNN successfully quantizes the weights and activations constrained to +1 or -1. Xnor-Net reduces the computation replacing 32-bit floating-point multiply accumulations by 1-bit xnor-popcnt operations. The above three articles still face a huge hurdle to put these methods in practice due to the limited accuracy on large-scale datasets such as ImageNet. In 2017, Zhou A et al.[181] improved the method with weights partition, groupe-wise quantization and achieved significant success in 5-bit quantization of AlexNet, VGG-16, GoogleNET. This approach employs weights that are either 0 or powers of 2, which allow multiplication to be implemented by bit shifts. In 2018, Benoit.J et al. [70] propose that an inference quantization scheme relies only on 8-bit integer arithmetic to approximate the floating-point calculation. This work is oriented on ARM NEON et reduce 4x of the model size without a drop of accuracy.

To conclude, the quantization error can't be ignored when quantizing the network into an extremely low bit (e.g., 2-bit or binary). The quantization noise in previous layers may be amplified to the next layer. However, quantization with Fine-tune/retraining demands added storage burden for complete datasets, which is not always feasible for the small hardware device. In contrast, quantization without training needs only a small set of calibration data.

4.4.1 Our quantization tool

A quantization tool is developed with a post-training method to quantize and organize the coefficient order before executing CNN inference as shown in figure 4.22.

The tool's inputs are floating-point coefficients extracted from any pre-trained model (*.h5) in the TensorFlow lib and ordered by the kernel. Since machine learning engineers know the impact of data width on resource usage by applying mathematical models, they

can determine the total data width. The tool searches the maximum integer part and then calculates the decimal amount. After quantization, these coefficients are sent to the CNN parameterized IP thru the serial link in the order of "bias followed by weight" and "full connected followed by convolution layer." The format of the input image in the inference still maintains a 16-bit fixed-point. The design flow of our quantization tool can be described as the following:

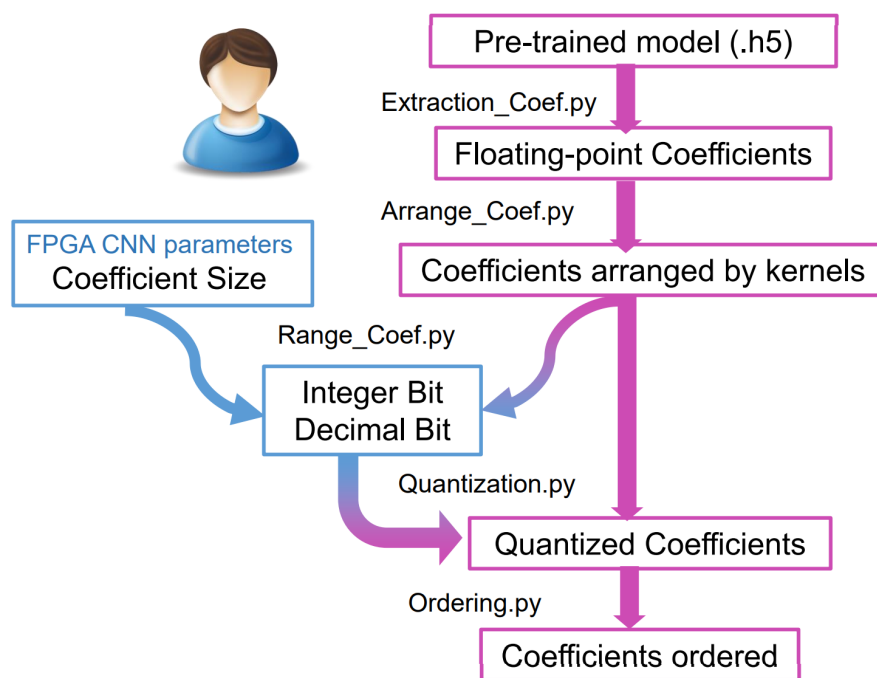


Figure 4.22 - Flow for quantifying and ordering coefficients of the CNN.

1. **Coefficients analysis:** This stage aims to find an adaptive fixed-point representation based on coefficients amplitude. The floating-point format assures a much larger dynamic range which is especially important when processing extensive data sets. In contrast, fixed-point representation has limitations when presenting a large range of data. Since weights and biases are represented in 32-bit floating-point in TensorFlow, hence coefficient analysis tool analyzes the amplitude of all coefficients for each layer to find an adequate range representation for the fixed-point formats.
2. **Bit-Width Reduction:** Quantization tool condenses the coefficients of a network to fixed-point format layer by layer. Given a network that contains a total of $\frac{N}{2}$ convolution and fully-connected layers with a combination of Q quantitative results (e.g., 16-bit, 14-bit, 12-bit), it takes up $\frac{N}{2}$ weights and $\frac{N}{2}$ biases. The computational

complexity is calculated by equation (4.13).

$$O(combination) = Q^N \quad (4.13)$$

After coefficient analysis, there are several methods to quantify these coefficients according to the data distribution. For example:

- Per-layer quantization, which means that the quantization is applied layer by layer, and each layer has a fixed-point representation;
- Network quantization, which means that the quantization is applied for the whole network, and each layer has the same fixed-pointed representation;
- Layer-regroup quantization, which means that the quantization is applied for the group of layers, and each group has a fixed-pointed representation.

We regroup convolution and fully-connected layers respectively according to their functionality in the networks, considering the computation complexity. Figure 4.23(a) presents the network quantization without regrouping, and Figure 4.23(b) shows a possible type of regrouping, both for LeNet-4. Table 4.5 lists two possible quantization sets for various sizes of fixed-point coefficients, from 6-bit to 16-bit.

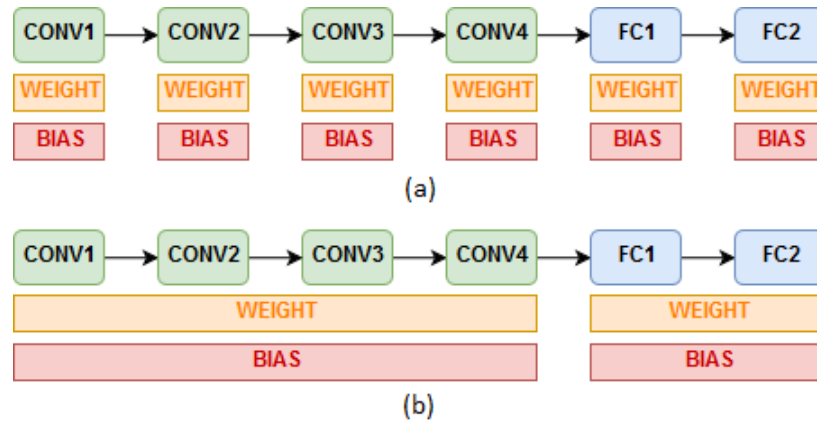


Figure 4.23 - Examples of no-regrouping (a) and a possible regrouping (b) in LeNet-4.

Table 4.5 - Number of combinations of regrouping and no-regrouping situations for LeNet-2 and LeNet-4.

CNN model	LeNet-2		LeNet-4	
Quantization (bits)	[16,14,12,10,8,6]	[16,8,6]	[16,14,12,10,8,6]	[16,8,6]
Complexity of Fig 4.23(a)	6^8	3^8	6^{12}	3^{12}
Complexity of Fig 4.23(b)	6^4	3^4	6^4	3^4

Regrouping coefficients can significantly decrease the number of combinations to test. The number of regrouping combinations is reduced by the factor of 3^8 compared to no-regrouping one in terms of a set of [16,8,6] for LeNet-4.

3. Test the Accuracy: All quantized weights and biases are updated in Tensorflow to execute the inference phase. The inference phase returns all accuracies for each quantization setting. All accuracies that exceed the given limited error tolerance will be ignored in the following stages.
4. Sizing data for CNNs blocks: The hardware library provides three basic hardware modules for CNNs: Convolution module, Pooling module, and fully-connected module described in VHDL. These generic modules can be used to automatically generate any CNN's structure with variable inputs: the input image size, the kernel size, the weight data width, and bias data width.

Consider a general signed fixed-point format with one signed bit for all modules: **fx m.n**, where m and n respectively refer to integer bits and fractional bits. Its range is $[-2^{m-1}, 2^{m-1} - 2^{-n}]$ with the numerical resolution 2^{-n} . Given a set of coefficients in regrouped layers, the quantization rule of fixed bits is :

- a) One bit is allocated to the sign;
- b) The minimum number of bits is allocated to the integer part according to the maximum coefficient extracted by the tool;
- c) The remaining bits are quantized for the decimal part. a) and b) are defined by **Data quantization tool** and c) can be specified by users with a different number of bits.

An example of 8-bit quantization can be found in the following figure:

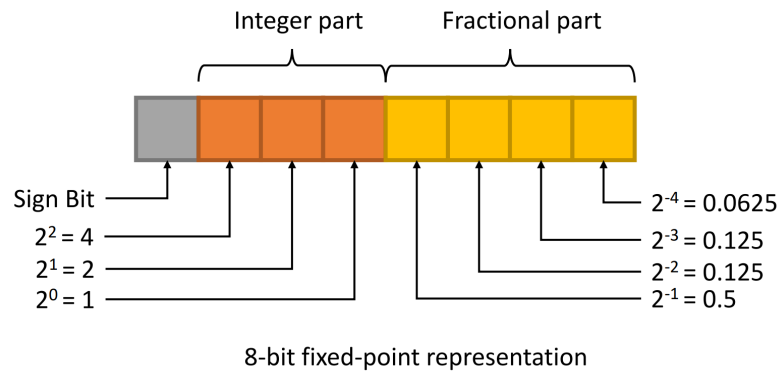


Figure 4.24 - Example of the 8-bit quantization.

5. FPGA resources analysis: Since the size of the network will be practically bounded by available memory, having an approximate value of Kbits required in a network is the prime task. This estimation is appropriate for all platforms(e.g., CPU, GPU, FPGA).

Sizing data tool developed in Python can evaluate FPGA resources used according to different quantization sizes of coefficients obtained in the previous stages as well as the type of CNNs implemented on FPGA.

4.4.2 Experimental results

This section investigates the effect of reduced bit-width for accuracy and resources in both convolution and fully connected layers. The proposed quantization sets are experimented on LeNet-2 using MNIST dataset and on LeNet-4 using CIFAR10 dataset.

4.4.2.1 Coefficient Analysis

Figure 4.25 enumerated weights and biases amplitudes of convolution layers and fully connected layers in LeNet4. As observed, the data amplitudes vary widely between -1 and 1 regarding different layers, which require a different number of bits to quantize each layer.

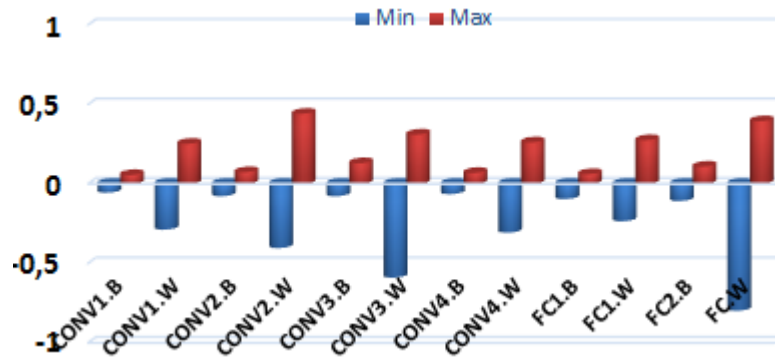


Figure 4.25 - Amplitudes of biases and weights of convolution layers and fully-connected layers in LeNet4

4.4.2.2 Test the accuracy

To explore the relationship between quantization bits and network performance, we experiment on regrouped layers in LeNet-2 and LeNet-4 with a set of [4,6,8,10,12,14,16] bits. A strong correlation is observed between the quantization number of convolution weights and accuracy in LeNet-4 (figure 4.26). This correlation analysis was repeated on

LeNet-2, and the quantization number of fully connected weights has a significant impact on the accuracy. Moreover, a further study on the impact of quantization number on

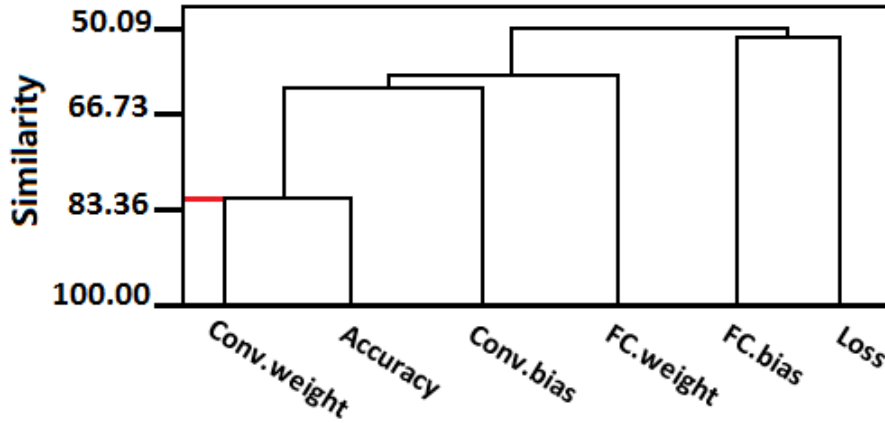


Figure 4.26 - Dendrogram of similarity among the variables of LeNet-4

accuracy was realized. Table 4.6 illustrates the result of quantized 32-bit floating-point LeNet-2 on the MNIST data set. The accuracy drops to 10% if Fully-Connected layer weights are represented with 4bit. In contrast, 4-bit quantization is optional for other coefficients while keeping high accuracy.

Table 4.6 - Accuracy of a quantization set for conv.biases, conv.weight, fc.biases, fc.weight.

Parameters		Quantized bits			
Convolution.weight	X ¹	X	4	X	X
Convolution.biases	X	X	X	4	X
Fully-connected.weight	4	X	X	X	X
Fully-connected.biases	X	4	X	X	X
Accuracy	10%	98% - 98.84%	93% - 96%	>98%	>98.5%

¹X represents a quantization bit from a set of 6,8,10,12,14,16 bit.

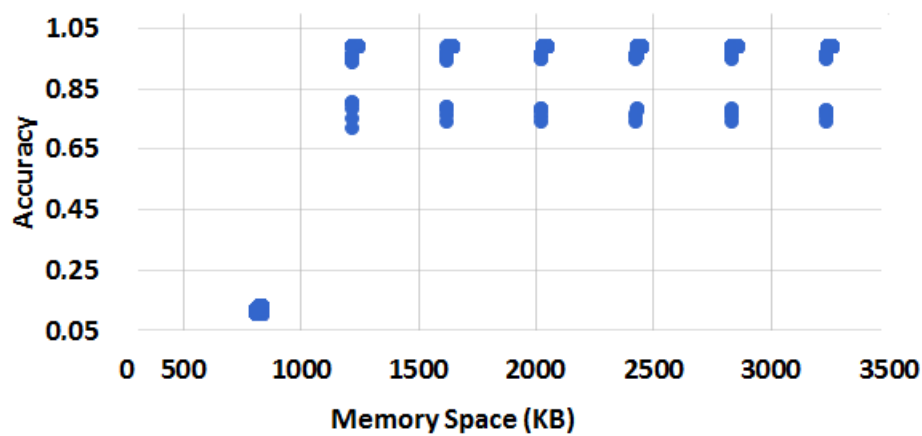
The possible quantization combinations with an accuracy higher than 80% on LeNet-4 are in the table 4.7. As indicated in the table, 6-bit for Conv. weight, Conv. bias and FC.weight and 8-bit for Conv. weight lead to a drop in accuracy (below 80%).

4.4.2.3 FPGA Resources Analysis

Figure 4.27 illustrates all quantization combinations of LeNet-2 located in *memory consumption - accuracy* axis. We can conclude that memory consumption varies widely on similar accuracy. It is possible to optimize memory space without losing accuracy. Besides, we evaluated the number of FPGA resources used by the CNN on Xilinx XC7V485T –

Table 4.7 - Number of quantizations in LeNet-4 with the accuracy more than 80%.

Quantization parameters	Eligible quantity			
	Conv.weight	Conv. biases	Fully-connected. weight	Fully-connected. biases
16-bit	97	96	101	71
14-bit	97	96	101	71
12-bit	111	96	105	72
10-bit	114	102	106	74
8-bit	0	29	6	67
6-bit	0	0	0	64

**Figure 4.27** - Memory - accuracy for All quantization combinations for LeNet-2

Vivado 2017.2. Consider a quantization set format: **X-X-X-X** bit where each X corresponds to the quantized bit for convolution weight, convolution biases, fully-connected weight, fully-connected biases. Table 4.8 lists the number of FPGA resources in LeNet-2. This

Table 4.8 - Approximate estimation of resources utilization in LeNet-2.

LeNet-2			
Quantization	Look Up Table	Flip-flop	DSP
8-4-8-4	309016	3627704	202416
14-12-10-8	382426	3755584	202416
16-16-16-16	609184	3960024	202416

analysis provides important insights into the association between quantization bit and FPGA resources. The quantized model with 8-4-8-4 bit and 14-12-10-8 bit combinations, which achieve higher than 98% accuracy, has greatly economized FPGA resources compared to resources used in 16-bit format network.

4.5 CONCLUSION

This chapter introduces the generation process of an FPGA-based general platform for deploying CNN. The platform is specially designed for machine learning engineers. From the perspective of machine learning engineers, all necessary IPs have been designed and verified. As a result, the CNN can be deployed on FPGA by machine learning engineers without hardware expertise. The machine learning engineers only need to define neural network parameters when generating the platform. Then, this chapter gives a detail of the required IP in the general platform. This chapter also proposes the optimization strategies and tools to obtain a well-performance platform.

In this chapter, we mainly introduce the usage of this platform to perform inference locally. However, with the rising attention of cloud computing, it has become a trend to place FPGAs in the cloud to enhance the computing power and resources capability. Therefore, in the following chapters, we propose a scenario for integrating the platform into a cloud environment. In this scenario, our platform can not only perform the inference but also involves more other usages.

PROPOSAL OF A FPGA-BASED CLOUD PLATFORM

Contents

5.1	OVERVIEW OF CNN-BASED APPLICATION LIFECYCLE	85
5.1.1	<i>Requirement analysis</i>	86
5.1.2	<i>Data Preparation</i>	87
5.1.3	<i>CNN design</i>	88
5.1.4	<i>CNN deployment</i>	88
5.1.5	<i>Evaluation</i>	89
5.1.6	<i>Production</i>	89
5.1.7	<i>Maintenance</i>	90
5.1.8	<i>Withdrawal</i>	91
5.2	REVIEWING CNN DEPLOYMENT TOOLS IN FPGA CLOUDS.	91
5.2.1	<i>Identified tools</i>	91
5.2.2	<i>Analysis of the tools according to the CNN application lifecycle</i>	92
5.2.3	<i>Challenges</i>	94
5.3	A FRAMEWORK FOR FPGA-BASED CNN DEPLOYMENT PLATFORM	97
5.3.1	<i>Contract manager</i>	99
5.3.2	<i>Architecture manager</i>	99
5.3.3	<i>Monitoring manager</i>	100
5.3.4	<i>Resource manager</i>	101
5.3.5	<i>Storage manager</i>	102
5.4	STUDY CASE	103
5.5	CONCLUSION	107

This chapter aims at designing an FPGA-based Cloud platform for executing DNN applications.

As early as 2014, many works such as [172][95] have investigated CNN implementations with diverse structures on FPGA. The millions of learnable parameters and billions of arithmetic operations lead to insufficient hardware resources compared to available FPGA resources. Several optimization techniques have been proposed to optimize the execution time or the number of resources, such as network quantization, network pruning, data path optimization. Two mainstream structures have been proposed: (i) streaming with a high pipeline, allowing concurrent executions, thereby achieving high throughput. However, as the deployed network is fully unrolling, numerous FPGA resources are exhausted for deeper networks, even if optimization strategies are adopted. Therefore, (ii) a single computation engine structure appears, which greatly economizes the FPGA resources but sacrifices the overall throughput.

With the increasing number of novel CNN models in the field of image classification, detection domain, many researchers (e.g.[173][51]) begin to develop frameworks that can automatically generate high-level synthesis or Register Transfer Level (RTL) architecture for diverse CNNs on the local FPGA. Due to the popularity of the FPGA Cloud, these frameworks have progressively become tools oriented to deploy CNN in FPGA Cloud. It makes excellent use of FPGAs in terms of available resources and performance and provides CNN implementation diversity. For example, [114] uses Xilinx SDAccel integrator and AWS integrator as backend and develops a framework that generates and integrates CNN architecture in the Cloud. [164] shapes DSPs as supertile units and are reused to accelerate CNNs, thus applying FPGA resources-as-service in the Cloud.

From these works, we can conclude that developing an FPGA-based CNN application has become crucial for their success. However, the design structure, the metrics, and techniques during the lifecycle are different according to various requirements of CNN applications. Meanwhile, it is unclear in the previous works what kinds of challenges and difficulties are faced in developing a CNN application on the FPGA. Therefore, it is vital to understand the software engineering practices of developing CNN applications on the FPGA to build valuable and effective techniques during the development. Unfortunately, little research has been conducted to summarize the critical point, the necessary elements, and expectations for developing an FPGA-based CNN application from the software engineering perspective.

In this chapter, we provide an overview to understand each phase of the FPGA-based CNN application during the lifecycle by making an in-deep analysis of the current works of CNN deployments on the FPGA. The main contributions are the following:

- We provide an overview of the FPGA-based application life cycle from the software engineering perspective. Also, the challenges and difficulties have been identified in all phases of the FPGA-based CNN application.
- We propose a novel architecture for the FPGA-based CNN platform, which aims at better covering all phases of this life cycle. Moreover, our platform provides more usages of CNN applications, which is not involved in the existing platforms.
- We also provide a method to construct a dedicated framework to achieve the multiple usages of CNN applications mentioned above in our platform. A special usage will be validated by this method in our paper to verify the feasibility of our proposed method.

The organization of the chapter is as follows: Section 5.1 presents the lifecycle of the FPGA-based CNN applications. In this section, we identify each phase and the necessary elements in the lifecycle. We also recognize the evaluation criteria for existing works in this section. Section 5.2 summarizes the characteristics of the existing frameworks and discusses the inconveniences and lacks these frameworks. Then, section 5.3 propose a novel architecture, which solves the problems in the existing frameworks. In addition, additional usage of the FPGA-based CNN applications has been submitted to meet the industrial and academic requirements. Finally, Section 5.4 shows the implementation results of our platform based on a particular problem.

5.1 OVERVIEW OF CNN-BASED APPLICATION LIFECYCLE

The CNN application framework provides a complete integration flow for the users to deploy several CNNs on the various FPGAs without hardware expertise. Furthermore, this framework is implemented in a cloud environment, aiming at sharing FPGA resources among multiple users to accelerate their CNN applications.

The lifecycle involves comprehensive and explicit steps to produce a CNN application that meets users' expectations within times and cost estimation. The lifecycle can give a defined view of the entire framework and verify the execution of each step in the frame-

work. The lifecycle consists of the detailed steps as shown in Figure 5.1: Requirement Analysis, Data Preparation, CNN Design, CNN Deployment, Evaluation, and Production. Each stage corresponds to a role or responsibility that the framework must understand, manage, and optimize to realize the CNN deployment in a cloud environment. The most crucial step is the CNN design through the entire lifecycle, as it involves selecting the hardware structure and the FPGA.

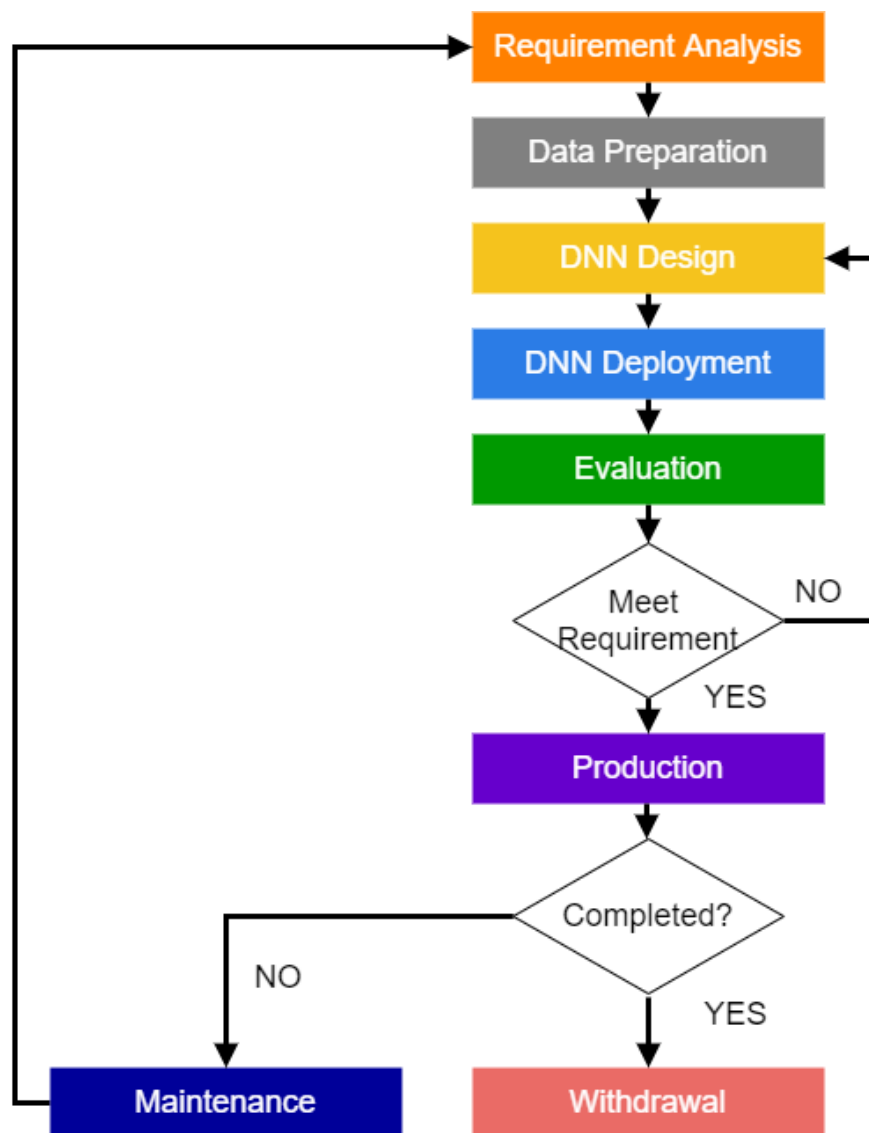


Figure 5.1 - Lifecycle of CNN-based application on the FPGA.

5.1.1 Requirement analysis

Requirement analysis is the process of collecting and analyzing the user's goals and correctly transforming problems into the requirements that the CNN-based application framework can handle. Requirement analysis is considered the essential stage in the

CNN-based application lifecycle since all decisions in subsequent stages highly depend on these requirements. The information extracted in the requirement analysis defines the usage scenarios of the CNN applications, thereby affecting the choice of CNN hardware structure or targeted FPGA in the following steps. In the CNN application process, a reasonable requirement analysis reduces risks of project abortion or delays and adds value to the analysis and design process. Therefore, the user's needs and conditions should be classified and identified to avoiding ambiguous requirements.

During the requirement analysis, the Usage of FPGA depends on the users' requirements. In the life cycle we proposed, we involved multiple usages in the requirements analysis to give a complete view of various requirements. To complete these usages, the user should specify a limited time, called Deadline. In the system's point, the Deadline refers to the time from the user's request submitted to the system until the system finally returns the desired result for the user. The waiting time in the system is busy processing other users' requests is also involved in the Deadline. Therefore, the response to the user's Deadline can be used as an indicator to measure the performance of the CNN-based application system.

Then, the user should identify several elements as the system's input that the system can process. Firstly, the user should declare the **Network type** (e.g., AlexNet, LeNet, LSTM). The system should support the selected network types by the user. The network type is an important element as it gives the complexity and the difficulty on the choice of the FPGAs and the hardware structure.

FPGA specifying is an optional element for users. According to different cases, users can select or not specify an FPGA device. The first case is that the user can specify a particular FPGA to complete the CNN execution. The second case is that the user can only describe the FPGA budget and FPGA family (e.g., ultra-scale), and the system will list all qualified FPGAs according to the user's needs. The last case is that the user does not specify any FPGA, and the system will choose with FPGA.

5.1.2 Data Preparation

Data preparation is the process of preparing datasets and coefficients for the training or inference and converting these data into a format that the framework can process. Data preparation is the essential since network learning or prediction is based on these data. Getting the correct data and data format can efficiently reduce the data loading workload.

Using poor quality data or poorly preparing data will result in unreliable output.

During the data preparation, it is necessary to identify the requirement for the dataset. For example, the user can locate the required bit data width to quantize the dataset and coefficients. Alternatively, the user can specify the batch size of the dataset.

5.1.3 CNN design

CNN design is the process of creating the CNN hardware model based on the hardware structures provided by the system. Therefore, CNN design relies on the requirement specifications produced in the first phase. Moreover, if the CNN application evaluation is invalid, the CNN design can also be re-produced to meet the evaluation requirements.

During the CNN design process, the **hardware structure** can be streaming structure or single-engine computing. The streaming structure has a pipeline structure, improving execution speed but consuming more hardware resources. Therefore, every time the requirements of the CNN change, the streaming structure should be recompiled to generate a new CNN design. On the other hand, single-engine computing is a fixed template and uses fewer hardware resources, increasing network execution time. In addition, if the CNN requirements change, single-engine calculations can implement a new CNN without hardware recompilation. These hardware structures can be described by **Programming model**, referring to the language or tool used to describe the hardware structure (e.g., HDL). In addition, **optimization** methods can be applied to optimize data paths and computing storage of CNNs, to adjust the CNN hardware model under resources or communication bound.

5.1.4 CNN deployment

CNN deployment is the process of implementing the generated CNN model in the **CNN design** process on the appropriate FPGAs. Furthermore, the CNN deployment translates the network requirements and data into physical components to meet the quality-of-service requirements. Therefore, the CNN deployment process depends not only on a solution proposed in the **CNN design** but also on performance and quality of service required.

During the CNN deployment, **deployment type** will be first determined by the user or the system according to the requirements. The deployment type refers to the number of CNN deployed on specific FPGAs, 1-CNN-to-n-FPGAs, n-CNNs-to-1-FPGA, n-CNN-

to-n-FPGA, or 1-CNN-to-1-FPGA. For example, the n-to-n mode is applied for the cloud environment, and n-to-1 can be used for the FPGA virtualization. Afterward, the system will select the appropriate FPGAs among all available FPGAs, in this context, namely **target FPGA**. The target FPGA(s) usually has enough resources to deploy CNN(s) and idle. According to the different situations, the selection of the FPGA varies:

- If the user initially specifies the FPGA device, the system will select the specified FPGA. If this FPGA is not accessible, the system will notify the user.
- If the user does not specify an FPGA device, the system will select several FPGAs to deploy the generated CNN model. Finally, the system returns a report with all possible results to the user.

The CNN deployment will be accomplished based on the **optimization condition**, which refers to the conditions under which the system stops optimizing the CNN hardware structure. For example, the system will control optimization if the system optimizes the hardware structure to meet resource requirements or throughput requirements.

5.1.5 Evaluation

Evaluation is a process of inferring an image or a small batch, estimating whether the result meets the requirements, and determining the subsequent process. The Evaluation is carried out in terms of the first process of the CNN-based application lifecycle: requirements analysis.

During the Evaluation, the processed results are compared to the initial requirements. In addition, the evaluation method contains several **metrics**, such as accuracy, throughput, to measure the functionality and effectiveness of the CNN-based applications. The metrics can help make decisions about the actions of re-designing CNNs or making designs into production. Moreover, **Aide-decision** including other conditions can help in understanding the limitations and drawbacks of the CNN application during the lifecycle and give a scope of the generated results.

5.1.6 Production

Production is the process of performing the inference on the complete dataset, processing the final results of the CNN application, and building out the CNN deployment in a

production environment. During the Production, the system will be evaluated by several criteria.

With the increasing efforts to provide a cloud environment for multiple tenancies to deploy CNNs on the shared FPGAs, resources, and performance **isolation** have become a concern in the cloud. CNN accelerators on the FPGA usually run under full hardware access and may share the same resources. Therefore, a malicious code can bring the attack on the whole platform for other tenancies [45, 163]. Also, dataset collection can be time-consuming and costly, especially in industrial cases where datasets are of great commercial value. Therefore, providing strict data and resource isolation for multi-tenants can prevent unauthorized access to the dataset and avoid data leakage [161]. Besides, a CNN application may make an impact on the performance of other CNNs applications during concurrent execution[45, 63], which causes unreliable performance. However, only a few works[170, 167] discuss the performance isolation problems, and the isolation is still under-explored.

Due to the complexity of CNN design, mapping CNN on FPGA requires specific hardware expertise, which is a long learning curve in a hardware description language (HDL) programming and performance optimization. Therefore, **productivity** has become a critical element in the design. Although high-level synthesis (HLS) provides the ease of designing CNNs for software engineers, it still requires basic low-level hardware knowledge to achieve good performance. Furthermore, according to the complexity of the CNN algorithm, deploying the CNN on the FPGA may be very time-consuming and may increase the programming burden of engineers.

Generally, the execution mode of CNN on the local FPGA is limited to a single user executing a single CNN within a given time. As a result, it remains difficult for a single local FPGA to support **multiple-tennacies** to perform numerous CNNs in parallel and meet each user's time, cost, and quality of service (QoS). Some frameworks (for example, [34]) successfully solve the problem of multiple CNNs scheduling but can only execute CNNs sequentially in the form of time slices in a single-task environment.

5.1.7 Maintenance

Assuming that the application has a problem in the production phase, maintenance is a process of upgrading, repairing the application, and fixing the problems. For complete coverage of the life cycle, we would have had to address the maintenance phase. The

maintenance stage requires that the application be evaluated to ensure that it does not become obsolete. Changes are made to the initial application by going back to a previous process. Maintenance enables an analysis of the software's performance and correctly resolving the issues that arise.

5.1.8 Withdrawal

We can assume two possibilities:

- the application progress successfully during the production phase, or
- the application had a problem in the production phase but was solved during the maintenance phase. Afterward, the application can successfully finish the production phase,

In these situations, withdrawal is the process of releasing occupied/reserved resources. Since these resources are large scale from lots of FPGA devices and with dynamical allocation during runtime, it is vital to release the resources as soon as possible to be allocated to other applications. Particularly, during the withdrawal phase, a resource scheduler should adjust the resources allocation or release to deploy new applications in the platform.

5.2 REVIEWING CNN DEPLOYMENT TOOLS IN FPGA CLOUDS.

This section aims to analyze the tools for deploying CNNs on FPGA clouds structured and systematic. We studied several works to identify the problems and challenges of developing a CNN application-based platform.

5.2.1 Identified tools

This part focuses on two types of tools: those that require local access to an FPGA and those that allow working with multiple FPGAs in the cloud environment.

In the local environment, [43] developed a framework for mapping the training on the FPGA cluster. The framework adopts a pipelined architecture with a one-dimensional topology. The authors evaluated their framework by training AlexNet on 10 Xilinx VC709 Connectivity Kits. [178] proposed a mapping method for implementing large-scale CNN applications across up to 16 FPGAs with resource and bandwidth limitations.

The method can partition the CNN application to each FPGA depending on the status of the FPGAs (busy or free). The estimation throughput of layer mapping depends on the FPGA topology, resource conditions, and neural network specifications. [173] proposed a hardware/software co-design framework to accelerate CNNs on the FPGA platform. The framework comprises a uniformed mathematical representation and a reconfigurable computation engine, which can generate different types of CNNs. The experiments are carried out on AlexNet and VGG16. The deployed VGG16 on the Xilinx VC709 and Xilinx KU060 can achieve 636 GOPS and 365 GOPs, respectively. [141] presents a domain-specific framework to automatically mapping CNNs into the FPGA platforms. The framework first translates a CNN description into the directed acyclic graph (DAG) and model the hardware platform with the specifications. Then, several transformations, such as graph partitioning, coarse- and fine-grained, are employed to optimize the design space efficiently. The experiments show that the framework can map LeNet-5, MPCNN, CNP while keeping high accuracy. Finally, [156] proposed an infrastructure based on the end-to-end compiler to optimize the CNN deployment on the FPGA. First, the infrastructure transforms a CNN architecture into a graph-level problem with the software description as input. Then, the infrastructure converts the description into directed acyclic graphs of computational operations.

In the cloud environment, [170] proposed a full-stack solution to virtualizing FPGA resources in a cloud for deploying CNNs. This solution supports resource sharing at both the node and multi-node levels. Furthermore, virtual blocks can be mapped across FPGAs to achieve timing closure and match communication delays by using a latency-insensitive interface. [167] proposed a framework aiming at providing isolation, sharing resources for the CNN task. The framework is applicable for any CNN accelerator in a cloud environment. Compared with previous methods, this technique solves physical resource isolation and performance among multiple users by sharing FPGA resources in the SDM method.

5.2.2 Analysis of the tools according to the CNN application lifecycle

We analyzed the characteristics of the reviewed tools in section 5.2.1 according to the stages of the lifecycle of CNN-based applications on the FPGA. The summary is as follows:

- Requirement analysis: The supported networks are usually CNN, which involves standard network operations like convolution, pooling, and fully connected. Other

networks such as RNN are generally not supported in these works as it requires a more complex design. Most of the usage of these works is "inference" and only one work supports the usage of "training" because these CNNs are applied for image processing, which contains the "training" and "inference" phases. The "training" is less achieved than the "inference" because it requires high precision for complicated mathematical operations. The listed works focus on deploying CNN on Xilinx FPGAs because Xilinx is the leader in FPGA manufacturing.

- **Data Preparation:** A few works have integrated quantification tools that can dynamically quantify the data width online in terms of data preparation. Without the quantization tool, the pre-trained coefficients always remain in floating-point format, which is impossible to adapt to any fixed-pointed form. Furthermore, the supported data width is usually 8 bit and 16 bit in the existing works since 8-bit or 16-bit will not significantly lose accuracy compared to the 4 bit or 2-bit quantization.
- **CNN design:** The works adopt two mainstream structures, single-engine, and streaming, to generate CNN hardware architectures. All other structures are derived from these two structures. The programming models are mostly high-level synthesis as it gains efficiency in development and is friendly for network optimization. We find that almost all works provide optimization strategies (e.g., looping and tiling) to economize resources usage. Without the optimization strategy, it is impossible to implement a whole network on the FPGA under the resources and bandwidth constraints.
- **CNN deployment:** All experimented FPGAs are usually the development boards that provides rich peripherals in the CNN deployment. Only a few works mentioned optimization conditions, that is, at which time, the tools finish design space exploration. Works concentrate on deploying one CNN/CNN to one or more FPGAs because multiple CNN/CNN deployments need to solve complex issues such as FPGA partial configuration or FPGA resource partition.
- **Evaluation:** In the evaluation, the metric "accuracy" is used to evaluate whether the deployed FPGA-based neural network maintains the same precision compared to the software-based network. If the accuracy drops a lot, we should modify the quantization strategy in the data preparation. "Throughput" is a standard metric to measure the instructional performance of the FPGA-based network, which is used

in all works. "Energy" is a metric to evaluate the energy consumption compared to GPU/CPU-based neural networks, and several works apply this metric to make an energy comparison of their deployed CNN. Finally, the aid-support focuses on the resources and bandwidth limitation as they are the two principal elements that limit the design deployment on the FPGA.

- **Production:** The listed works cover the "production" step to perform the CNN inference on the produced hardware architecture finally. However, these works mostly ignore the crucial evaluations, such as "isolation", "productivity", and "multi-tenancies".

The works identified in the literature are still experimental. This lack of maturity of the tools means that most downstream life cycle stages are not discussed. Therefore, we find none of the existing works cover the entire life cycle described in section 5.1.

5.2.3 Challenges

Improved coverage of the use case: The mentioned works only provide the inference and training. From a more general perspective, machine learning engineers may have other purposes when using the platform. To achieve these purposes with maximizing QoS criteria (e.g., execution time, energy consumption, and financial costs), machine learning engineers can select functions provided in the framework. Then, the framework will return different results according to the selected functions. We summary three use cases that can describe functions used for different purposes in the framework: Image processing, Network parameterization, or Model exploration shown in the figure 5.2.

- *Image processing:* This use case refers to infer CNN on the dataset provided by the Machine learning engineer. ML engineers must provide the network model(or network configurations), network data parameters(specific bit width), and pre-trained coefficients.
- *Network parameterization:* The ML engineer knows the CNN model but cannot determine the network data parameters (bit width). Our framework will explore various bit widths so that ML engineers can study the impact of data bit width on classification accuracy and select appropriate parameters.
- *Models exploration:* The ML engineer only has the data set to be trained or tested. He does not know the network model to be used and related network data parameters.

Table 5.1 - Several works which develop the tool of deploying CNN applications on the FPGA.

Works	Requirement analysis			Data preparation			CNN Design			CNN deployment			Evaluation			Production			
	Network type	Training	Usage	FPGA family	Quantization tool	Data width	Hardware structure	Programming model	Optimization techniques	Targeted FPGA	Optimization condition	Deployment type	Accuracy	Throughput (GOPS)	Energy	Aide-decision	Isolation	Multi-tenancy	Productivity
[43]	CNN	YES (16 bit fixed)	Training	Catapult (xilinx Virtex6)	Not integrated	N/M	Streaming	HLS	Loop tiling, unrolling	Catapult (xilinx Virtex6)		1 CNN to N FPGAs	YES	YES	YES	Available FPGAs in the cluster and resources limitation	N/A	NO	YES
[178]	Large-Scale CNN	NO	Inference	Xilinx	Not integrated	16 bit fixed	N/M	HLS	Loop tiling, unrolling, Parallelism	Xilinx Ultra-Scale	Binary search until satisfying resource	1 CNN to N FPGAs	YES	YES	NO	Resource and bandwidth limitation	N/A	NO	YES
[173]	CNN/CNN	NO	Inference	Xilinx	YES	8/16 bit fixed	Single engine	HLS	Loop tiling, unrolling	Xilinx		1 CNN to 1 FPGA	YES	YES	NO	bandwidth limitation	N/A	NO	YES
[141]	CNN	NO	Inference	Xilinx	Not integrated	N/M	Streaming	HLS	Loop tiling, unrolling	Xilinx ZC706	Optimise until satisfying resources and latency requirements	2 CNN to 1 FPGA	YES	YES	YES	Resource limitation	N/A	YES	YES
[?]	CNN	NO	Inference	Xilinx	YES	any data-width	Single engine	HDL	Loop tiling, unrolling	Xilinx ZU9 and ZU2	1D feature maps can be stored on-chip. 2D dependency among operations is pre-defined in the templates	1 CNN to 1 FPGA	YES	YES	YES	Resource limitation	N/A	YES	YES
[170]	CNN/CNN	NO	Inference	Xilinx	YES	8 bit fixed	Single engine	HLS	Loop tiling, unrolling	Xilinx		N CNN to 1 FPGA	YES	YES	YES	Resource and bandwidth limitation	YES	YES	YES
[167]	CNN/CNN	NO	Inference	Xilinx	YES	N/M	Single engine	HDL	Loop tiling, unrolling	Xilinx Alveo	Tiling with minimal latency	1 CNN to 1 FPGA	YES	YES	YES	Resource and bandwidth limitation, reconfiguration overhead, latency	YES	YES	YES

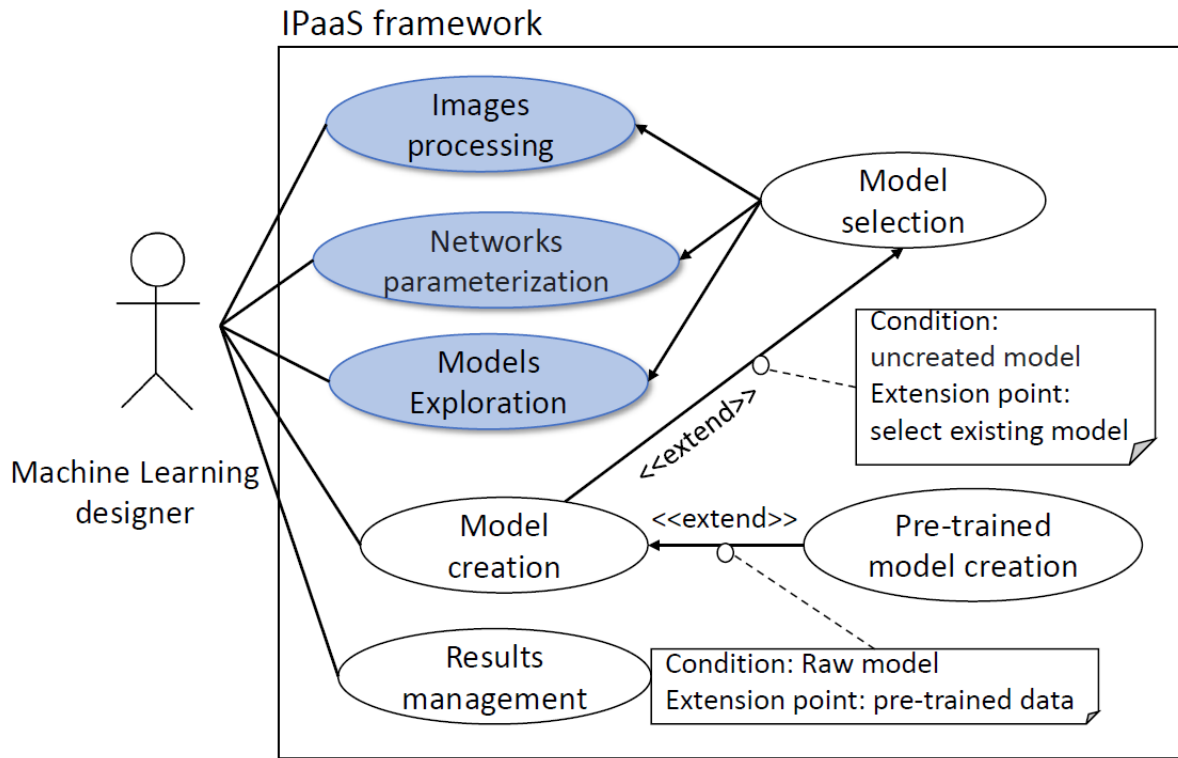


Figure 5.2 - Use cases and main use cases (marked in blue) in the framework

Therefore, the framework will use the provided data set to explore different CNN models and define the most suitable CNN architecture on the FPGA.

Completed lifecycle for CNN applications: These works did not discuss the complete process from generating CNN architecture to selecting the appropriate FPGA to finally executing the use case in the cloud.

Moreover, the platform's lifecycle proposed by these works is not complete. There is a lack of "maintenance" and "withdrawal" steps to ensure the execution of the application on the platform. Finally, the metrics of these works are not discussed, which are the crucial part of evaluating the platform's performance.

The maintenance and withdrawal phases should be addressed for complete coverage of the life cycle. The maintenance stage requires that the application be evaluated to ensure that it does not become obsolete. Changes can be made to the initial application, which translates into the life cycle by going back to a previous stage. The withdrawal phase is also critical for a complex system. The framework manages the dynamical deployment of

the applications on FPGA clusters, and resources can be added and removed at runtime through the withdrawal step.

Without hardware expertise: Several reviewed works require the hardware expertise for machine learning engineers to implement CNN on FPGAs (from converting CNN algorithms to hardware descriptions and finally implementing CNN on FPGA devices).

In order to solve the problem of machine learning engineers deploying CNN without hardware expertise, it is necessary first to develop several hardware IPs that can be used to generate different CNN structures proposed by machine learning engineers. Then, a method for modeling these IPs based on the resource conditions of different FPGA devices should be proposed. Finally, a framework that integrates different agents should be developed to manage and control all the necessary processes during the CNN deployment.

5.3 A FRAMEWORK FOR FPGA-BASED CNN DEPLOYMENT PLATFORM

This section proposes a framework whose usage is based upon a new type of FPGA Cloud dedicated to machine Learning engineers who have no hardware expertise in FPGA design, IP design, and FPGA implementation. Furthermore, our framework integrates multiple managers, combined with mapping the CNN model on appropriate FPGAs in the Cloud.

Inspired by the works [77, 147], we proposed an overall system architecture of our platform, which is essentially composed of six server-side managers and necessary modules to construct the platform as shown in the figure.

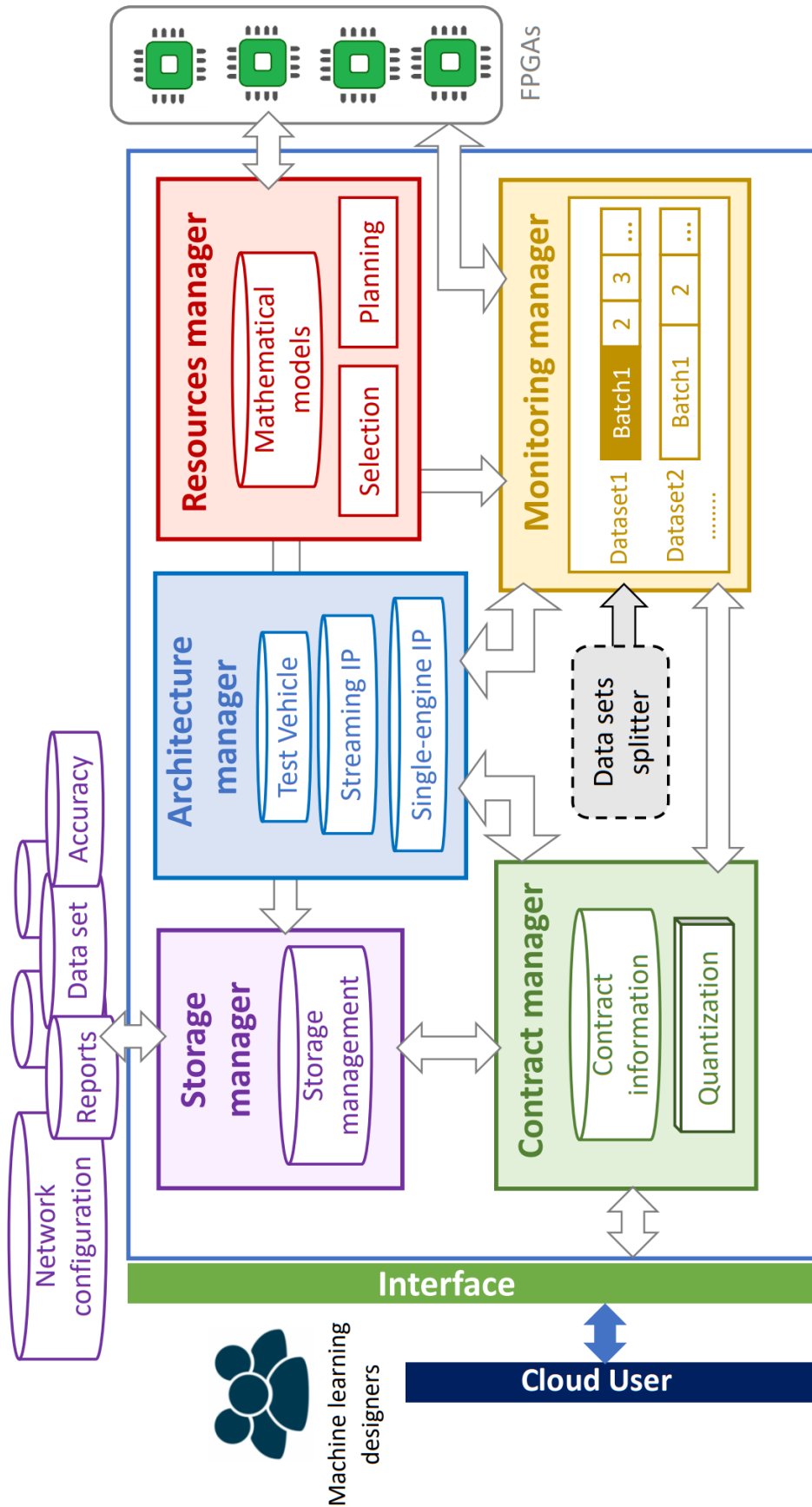


Figure 5.3 - An overview of the framework for FPGA-based CNN deployment platform in the cloud

5.3.1 Contract manager

The contract manager performs the contract's life cycle management, from the submission of the contract (e.g., data, QoS, network configuration) to the completion of the network execution within the deadline. It also provides an interface for the ML engineer to interact with the framework. In our context, the contract guarantees the requirements (QoS) of the ML engineer in the "client" use case mentioned in the previous section.

A coordination mechanism is also provided for contract managers to avoid conflicts between submitted contracts. If the contract manager encounters difficulties during contract execution, the contract manager can negotiate a coordination mechanism with other contract managers. For example, suppose other contracts already occupy the appropriate FPGA selected in agreement A for a period P . In that case, the contract A manager will negotiate with other contract managers. Then, the contract A manager jumps in the queue to use these FPGAs or divides period P into several periods and then deals with other managers again.

The contract manager also manages the quantization tool that converts data from floating-point to fixed-point and offers the data size configurations: integer part D_{int} , fractional part D_{dec} , total bit width D . Thus, the machine learning engineer can specify the integer and decimal parts of coefficients or select the total number of bits. The integer and fractional parts are automatically decided.

5.3.2 Architecture manager

The architecture manager mainly uses the mathematical models to select the appropriate CNN architecture and generates a bitstream of the CNN hardware architecture (including the test vehicle). Then, the bitstream is downloaded on the FPGA, decided by the resource manager towards the machine learning use cases.

The architecture manager copes with the available CNN IPs and the test vehicle. The CNN IPs are two types of templates that can generate CNN architectures are presented in the Cloud framework.

- **Streaming:** This is a CNN data flow through a chain of sequential CNN IPs. It is a general-purpose structure that can be configured into several networks according to the hardware description language (HDL) package parameters. The streaming architecture is generated according to the model and parameters provided by the

machine learning engineer. The CNN architecture is developed for each contract using CNN streaming IPs available in the IP library. The framework generates the CNN architecture with the associated test vehicle, evaluates the required resources of the final architecture using mathematical models, and then selects the FPGA used to download the bitstream. Several mathematical models depend on the FPGA families and providers, which evaluate FPGA resource usage under a given network configuration and model. This structure is mainly used to infer the machine learning engineer's database on a parameterized CNN model (Use Case presented in the previous section).

- **Single computation engine:** This is a fixed structure with a high degree of reconfigurability that has already been implemented on FPGA devices. This structure is mainly used for the use cases of Model exploration and Network parameterization. In addition, it can develop part of the functions of CNN functions, such as convolution, softmax, LSTM. The processor in CPU-FPGA heterogeneous architecture sends instructions to configure and reuse this computation engine multiple times to deploy a large and complex network in the cloud.
- **Test vehicle:** The CNN IPs are connected to this test vehicle inside the FPGA. The test vehicle is considered an IP defined by STMicroelectronics, which is used to transmit data from FPGA and extract data from CNN IPs. More precisely, the CNN coefficients and datasets stored in the cloud are sent by the storage manager to the external memory of the FPGA and then sent to the CNN IP through the test vehicle. Finally, the execution results are sent back from CNN IPs to the memory using the test vehicle. Thus, the test vehicle integrates multiple communication infrastructures to achieve the required bandwidth. In our framework, figure 5.4 and figure 5.5 describe both types of test vehicles used for the single-engine and streaming structure.

5.3.3 Monitoring manager

The monitoring manager splits datasets into multiple batches according to the task queue and maintains queues of images waiting to be processed. It also supervises and controls the execution of CNNs on FPGAs. If an error or a failure occurs in the framework, it can notify the machine learning engineers through an early warning mechanism. The monitoring manager keeps the execution of the application on track, and they can dynamically

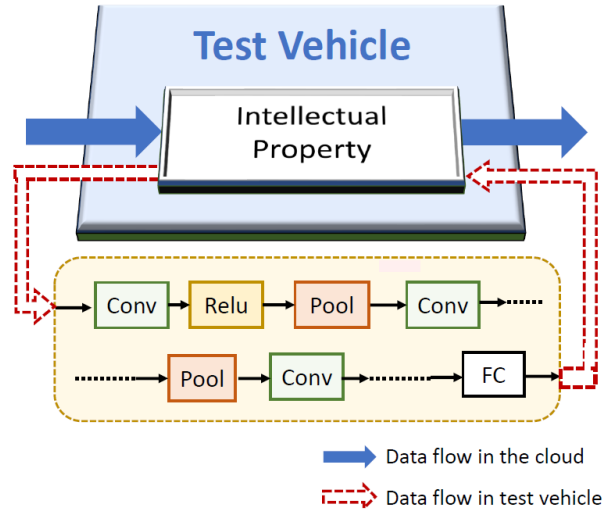


Figure 5.4 - Streaming (including the data flow) mounted on the test vehicle

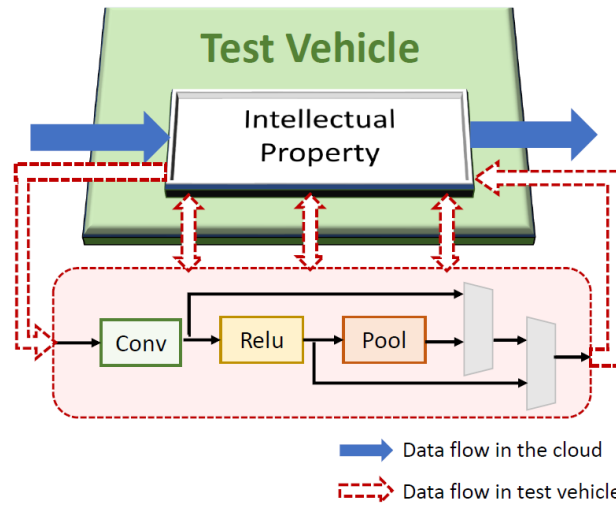


Figure 5.5 - Single computation engine (including the data flow) mounted on the test vehicle

adjust the state according to the current situation. For example, if the execution time of the current application is longer than expected, the monitoring manager will notify the contract manager to negotiate with other contract managers.

5.3.4 Resource manager

The resource manager is basically in charge of all FPGAs' scheduling and usage, including a real-time list of all unused or unreserved FPGAs and selecting a suitable FPGA among all these FPGAs to meet the requirements in the contract. The FPGA selection includes but is not limited to the number of resources available on the FPGA, the family of the FPGA, the state of the FPGA (e.g., the occupied state after the period $P1$, the idle state, or

the idle state).

The resource manager manages the mathematical models. The framework can automatically generate the hardware architectures from CNN IPs. This automatic generation relies on the FPGA resource usage model, which estimates the FPGA resource usage of a CNN hardware architecture without synthesis and place & route steps for the architecture and resources managers. The mathematical models also provide optimal mapping strategies under restricted FPGA resources.

This mathematical model can maximize the use of resources inside the FPGA and freely constraints the type of resource used. Thus, it is possible to deploy a part of the computationally intensive operation from DSP resources to LUT to accommodate deeper layers and efficiently use the available FPGA resources.

5.3.5 Storage manager

The storage manager ensures that the data can be completely and safely stored for a period $P2$ in the stock without a lack of information. In addition, it provides an interface for the monitoring manager and contract manager, which can efficiently and flexibly access large amounts of data.

The storage manager stores network configurations, which are used for the machine learning engineer to describe CNNs, including filter size K , stride size S , padding size Ps , pooling size P , input and output feature maps size Fi and Fo , number of layers N_{layer} , with or without activation function $Valid_A$, with or without pooling function $Valid_P$.

The storage manager also stores the CNN dataset and processed results. The processed results can be:

- The accuracy of the CNN. This result is obtained in the use case "Image processing";
- A report of resources utilization for the application. This report is obtained in the use case "Network parameterization". The machine learning engineer will parameterize the CNN structure with different data widths. Therefore, the framework returns a report of resource utilization about these data widths;
- A report of the feasibility. This report is obtained in the use case "Model exploration". The machine learning engineers will test several CNN structures under the FPGA resources conditions. So the framework will return a report to identify the feasibility of each CNN design.

5.4 STUDY CASE

In this part, we demonstrate the manager's workflow based on the use case "image processing". Once the machine learning provides the network model and QoS to the cloud framework, the contract manager will request other managers to undertake their tasks. These managers communicate and collaborate to complete a manager-based scheduling mechanism. Finally, the framework can use this mechanism to automatically complete the deployment of CNN on FPGA in the cloud.

The figure 5.6 illustrates a sequence diagram of the use case. The details can be explained as follows:

- Precondition of the use case:

The Machine Learning Engineer knows:

1. The CNN model he wants to implement;
2. The CNN parameters he sized on his model;
3. The coefficients of the model;
4. The dataset;
5. At least one FPGA exists in the framework.

- Steps of the use case:

1. The machine learning engineer logs in by giving the identification information;
2. The system authorizes the connection;
3. The machine learning engineer selects the CNN model;
4. It specifies the values of the parameters of the model;
5. It specifies the values of the parameter coefficients of the model;
6. It specifies QoS criteria (due date, criteria hierarchy);
7. The system determines the structure of the streaming / single-engine model;
8. The system generates the hardware architecture of CNN;
9. The system determines the FPGAs that can contain the CNN hardware architecture from among the FPGAs (i.g., the appropriate FPGAs);
10. The system elects the FPGA from the appropriate FPGAs and immediately available;

11. The system implements the CNN on the elected FPGA;
 12. The system informs the Machine Learning Engineer that the CNN has been installed on the FPGA;
 13. The machine learning engineer sends images;
 14. The system makes inference based on images;
 15. The system produces the result of processing its image database (classified images) and an execution report (accuracy, CNN parameters, coefficients) to the machine learning engineer;
 16. The system notifies the Machine Learning engineer the result;
 17. The system releases the FPGA.
- Postcondition of the use case:
The results (all classified images or CNN details) have been produced.
 - Exception of the use case: No suitable FPGA:
The scenario starts at step 9;
The system replaces steps 6-10 with:
 7. The system cannot find a suitable FPGA;
 8. The system generates an alert for the platform manager;
 9. The system informs the machine learning engineer that the CNN cannot be installed on the FPGA;
 10. End of the exceptional scenario.
 - Alternative 1 of the use case: Delay after the FPGA reservation deadline
In step 10, there is no FPGA available that meets the deadline.
The system adds a step:
 10. The system informs the machine learning engineer that no suitable FPGA can meet the deadline;
 11. The system asks the user to enter a new deadline Return to step 6 of the nominal scenario.
 - Alternative 2 of the use case: Reservation of an FPGA available not immediately but for a sufficient period before expiry.
In step 10, no immediately available FPGA meets the deadline.
The system adds a step:

11. The system reserves an FPGA on date d for a sufficient period for processing.
 12. The system informs the user of the date on which the result will be available
- Return to step 6 of the nominal scenario.

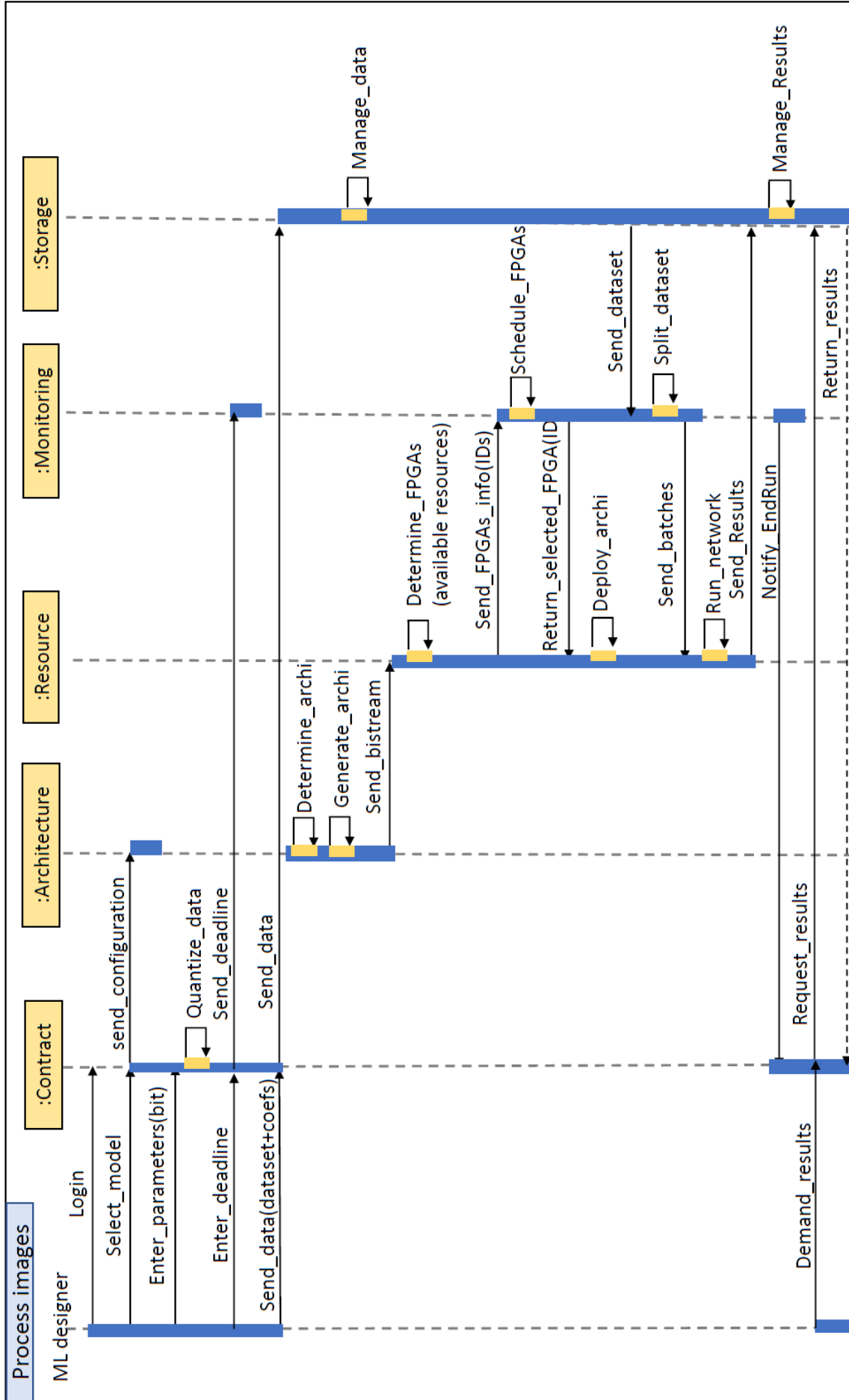


Figure 5.6 - The sequence diagram of the "Image processing" use case

5.5 CONCLUSION

Since IP design and implementation on the FPGA remains a primary challenge for the machine learning engineer, we provide a methodology to address this problem.

We first provide the life cycle of FPGA-based DNN applications and identify the main difficulties during CNN deployment. Then, we proposed a framework designed to provide machine learning engineers with various functions under different QoS. The proposed framework puts CNN IP design, quantization tools, and FPGA resources in the cloud and integrates multiple managers to generate a suitable CNN architecture and select FPGA. With such a framework, machine learning designers can infer and explore CNN without any hardware expertise. Finally, this chapter explains the workflow of a use case to demonstrate the feasibility of our proposed framework.

CONCLUSION AND PERSPECTIVES

Contents

6.1 CONCLUSION	110
6.2 PERSPECTIVES	110

This chapter concludes this thesis with, first, a synthesis of the work carried out and the results obtained. Then in a second step, a list of proposals for improvements and further research is proposed.

6.1 CONCLUSION

In this thesis, we first give an insight into the technology used by the FPGA-based platform for CNN deployment and extract the trend of such an FPGA-based platform from local to the cloud. At the same time, we identify the challenges of this local-to-cloud evolution, which motivates us to propose a new platform suitable for the cloud environment.

The proposed platform is dedicated to machine learning engineers executing neural networks on FPGA Cloud without hardware expertise. The platform integrates two types of CNN hardware IP, generating different types of networks according to resource usage and execution time requirements. The platform also includes a quantization tool for data width optimization. In addition, the integrated mathematical model in the platform can estimate the resource utilization of the generated network. Therefore, machine learning engineers can select and design networks to be executed on FPGAs without synthesis tools.

To integrate the platform into the cloud environment, we proposed the life cycle of an FPGA-based CNN application, which contains all the necessary steps. At the same time, we have also identified the challenges and difficulties of each step in the life cycle. Finally, based on the refined life cycle, we have added more use cases that can be executed for machine learning engineers and provided a way to build special tools to achieve multiple uses of the CNN mentioned above applications in our platform.

6.2 PERSPECTIVES

The Network on Chip (NoC) is a communication infrastructure whose goal is to facilitate the interconnection between IPs. Integrating CNN IP on Noc has the following advantages:

- Scalability as CNN IPs can be flexibly connected to the bus;
- Shared communication path between CNN IPs which efficiently reduces resources usage;
- Independence between data processing and communication.

Putting the DNN IP in the NoC will not reduce the performance of the implementation because it handles the problem of a large number of fan_out existing in point-to-point connections, and a large number of fan_out will seriously reduce the results (resources and time) of the implemented DNN.

Publications

JOURNAL

Chen Wu, Virginie Fresse, Benoit Suffran, Hubert Konik, Accelerating DNNs from local to virtualized FPGA in the Cloud: A survey of trends, Journal of architecture system 119 (2021).

INTERNATIONAL CONFERENCES

Chen Wu, Virginie Fresse, Benoit Suffran, Hubert Konik, A survey on Deep Neural Network accelerators: From local to virtualized FPGA in the Cloud, The 12th International Conference on Information, Intelligence, Systems and Applications (IISA).

Chen Wu, Virginie Fresse, Benoit Suffran, Hubert Konik, Mathematic models based on multiple-criteria decision analysis for tuning industrial CNN in an FPGA computing cluster, The 31st International Workshop on Rapid System Prototyping (RSP).

NATIONAL CONFERENCES

Chen Wu, Virginie Fresse, Benoit Suffran, Hubert Konik, Accélération de l'inférence d'un CNN paramétrable sur des composants FPGAs déployés dans un cluster. GDR ISIS Cluster SoC HPC, 2020.

List of Figures

1.1	Cloud computing-type platform for CNNs inference on the FPGA.	4
2.1	Structure of a convolution neural network[6].	8
2.2	Example of the convolution operation.	9
2.3	Example of the training process	12
2.4	Xilinx Zynq-7000 block diagram[155].	15
2.5	FPGA design flow.	16
2.6	General method for the CNN.	19
3.1	Characteristics of deploying FPGAs in the cloud and FPGA virtualization for CNN deployment.	24
3.2	IaaS and SaaS FPGA cloud. "Vendor manage (optional)" and "User manage (optional)" indicate that this hierarchy does not always exist in the FPGA cloud, and it is customised by each FPGA cloud vendor or user.	27
3.3	Overall architecture of the FPGA-based CNN accelerators in the IaaS cloud. (a) Different levels of abstraction in the FPGA virtualization technique. (b) Example of the system architecture in node-level virtualization.	29
3.4	Example of accelerating CNNs using the streaming architecture.	30
3.5	Example of accelerating CNNs using the single computation engine accelerator architecture.	31
3.6	Generic frameworks for CNN accelerators.	34
3.7	Generic compiler-inspired frameworks for CNN accelerators.	34
3.8	(a)CNN deployment without virtualization. (b) Example of FPGA virtualization at the node level for deploying one CNN on a local FPGA. (c) Example of FPGA virtualization at the node level for deploying several CNNs in the cloud environment.	36
3.9	Evolution of CNN accelerators at each time node: from manual mapping to frameworks, from a single node to a cluster, from physical to virtual resources, and from local to cloud.	43
3.10	Comparison of related methods with different characteristics.	45
4.1	Architecture of our platform.	50
4.2	Conception flow of our platform.	52
4.3	The global structure of the FPGA-based platform.	53
4.4	Streaming CNN IP with AXI4-Streaming.	54
4.5	Global design of the streaming CNN IP in Xilinx Vivado.	55
4.6	Data flow of extract neighborhood pixels in with line buffer.	56
4.7	Data flow of loading pixels in point-to-point mode.	57
4.8	Data flow of loading coefficients in serial link.	58
4.9	Principle of weights and bias ordering.	58
4.10	Method for constructing the mathematical models	59
4.11	Effect of data width on Logic LUT and DSP.	61
4.12	Mathematic models of LUT Memory used in terms of Layer1	63
4.13	mathematic models of LUT Logic Used in terms of Layer2	64
4.14	Mathematic models of flipflop Used in terms of Layer2	64
4.15	Mathematic models of Logic LUT in terms of Layer3.	65
4.16	Mathematic models of Flipflop in terms of Layer3.	65
4.17	Mathematic models of FPGA resources in terms of FC1.	66
4.18	mathematic models of FPGA resourceS in terms of FC2	67
4.19	Single engine CNN IP with AXI4-full standard.	68
4.20	Global design of the single-engine CNN IP in Xilinx Vivado.	70
4.21	Different quantization methods.	74
4.22	Flow for quantifying and ordering coefficients of the CNN.	76

4.23	Examples of no-regrouping (a) and a possible regrouping (b) in LeNet-4.	77
4.24	Example of the 8-bit quantization.	78
4.25	Amplitudes of biases and weights of convolution layers and fully-connected layers in LeNet4 . . .	79
4.26	Dendrogram of similarity among the variables of LeNet-4	80
4.27	Memory - accuracy for All quantization combinations for LeNet-2	81
5.1	Lifecycle of CNN-based application on the FPGA.	86
5.2	Use cases and main use cases (marked in blue) in the framework	96
5.3	An overview of the framework for FPGA-based CNN deployment platform in the cloud	98
5.4	Streaming (including the data flow) mounted on the test vehicle	101
5.5	Single computation engine (including the data flow) mounted on the test vehicle	101
5.6	The sequence diagram of the "Image processing" use case	106

List of Tables

2.1	Standards and interconnections between IPs.	16
2.2	Difference between AXI4-full, AXI4-lite, AXI4-streaming.	17
3.1	Comparison of streaming and single computation architectures for CNN acceleration.	32
3.2	Several examples of manual mapping and frameworks on the local FPGA.	35
3.3	Comparison of coarse- and fine-grained overlays on FPGAs for CNN acceleration	37
3.4	Several examples of CNNs based on local virtualized FPGAs and CNNs in the cloud.	41
4.1	Pearson's correlation (>0.7) extracted from R.	62
4.2	Configuration of each CNN	67
4.3	Timing evaluation of loading coefficients and performing inference.	67
4.4	Comparison of latency results (clock cycle) under different directives.	73
4.5	Number of combinations of regrouping and no-regrouping situations for LeNet-2 and LeNet-4.	77
4.6	Accuracy of a quantization set for conv.biases, conv.weight, fc.biases, fc.weight.	80
4.7	Number of quantizations in LeNet-4 with the accuracy more than 80%.	81
4.8	Approximate estimation of resources utilization in LeNet-2.	81
5.1	Several works which develop the tool of deploying CNN applications on the FPGA.	95

Bibliography

- [1] Amazon ec2 f1 <https://aws.amazon.com/fr/ec2/instance-types/f1/>.
- [2] Huawei acceleration cloud server(facs).
- [3] Imagenet ilsvrc challenge, <http://www.image-net.org/challenges/lsrvr>.
- [4] Intel® stratix® 10 variable precision dsp blocks user guide.
- [5] Microsoft catapult <https://www.microsoft.com/en-us/research/project/project-catapult/>.
- [6] Structure of convolution neural network, <https://fr.mathworks.com/discovery/convolutional-neural-network-matlab.html>.
- [7] Wishbone interconnection architecture, <https://cdn.opencores.org/downloads/wbspecb3.pdf>.
- [8] Xilinx vitis ai. <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>.
- [9] Mohamed S Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O’Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C Ling, et al. Dla: Compiler and fpga overlay for neural network inference acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 411–4117. IEEE, 2018.
- [10] Kamel ABDELOUAHAB, Maxime Pelcat, Jocelyn Sérot, Cédric Bourrasset, and François Berry. Tactics to Directly Map CNN graphs on Embedded FPGAs. *IEEE Embedded Systems Letters*, 9(4):113 – 116, 2017.
- [11] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. Reconos: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2014.
- [12] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, and Tobi Delbruck. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Transactions on Neural Networks and Learning Systems*, 30(3):644–656, 2019.
- [13] Alibaba. cloud,<https://eu.alibabacloud.com/>.
- [14] Altera. Altera avalon, <https://www.intel.com/content/dam/manual/mnlavalonspec.pdf>.
- [15] Altera. Altera axi core, <https://www.intel.com/content/www/us/en/programmable/support>.
- [16] Amazon. Aws, <https://aws.amazon.com/fr/>.
- [17] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1131–1135. IEEE, 2015.
- [18] Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy K. John. Tensor slices to the rescue: Supercharging ml acceleration on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’21*, page 23–33, New York, NY, USA, 2021. Association for Computing Machinery.

- [19] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy, and Paolo Ienne. Virtualized execution runtime for fpga accelerators in the cloud. *IEEE Access*, 5:1900–1910, 2017.
- [20] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems*, pages 7948–7956, 2019.
- [21] Jeff Barnes. Azure machine learning. *Microsoft Azure Essentials*. Microsoft, 2015.
- [22] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.
- [23] Ahmed Ghazi Blaiech, Khaled Ben Khalifa, Carlos Valderrama, Marcelo A.C. Fernandes, and Mohamed Hedi Bedoui. A survey and taxonomy of fpga-based deep learning accelerators. *Journal of Systems Architecture*, 98:331–345, 2019.
- [24] Michaela Blott, Thomas B Preußner, Nicholas J Fraser, Giulio Gambardella, Kenneth O’Brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, 11(3):1–23, 2018.
- [25] Thierry Bouwmans, Sajid Javed, Maryam Sultana, and Soon Ki Jung. Deep neural network concepts for background subtraction: A systematic review and comparative evaluation. *Neural Networks*, 117:8–66, 2019.
- [26] S. Byma, J. G. Steffan, et al. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116, 2014.
- [27] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. Cloud-dnn: An open framework for mapping dnn models to cloud fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’19*, page 73–82, New York, NY, USA, 2019. Association for Computing Machinery.
- [28] S. Alexander Chin, Kuang Ping Niu, Matthew Walker, Shizhang Yin, Alexander Mertens, Jongeun Lee, and Jason H. Anderson. Architecture exploration of standard-cell and fpga-overlay cgras using the open-source cgra-me framework. In *Proceedings of the 2018 International Symposium on Physical Design, ISPD ’18*, page 48–55, New York, NY, USA, 2018. Association for Computing Machinery.
- [29] Yoni Choukroun, Eli Kravchik, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. *CoRR*, abs/1902.06822, 2019.
- [30] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Hussein, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bitu Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving dnn in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.
- [31] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.
- [32] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.
- [33] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated fpgas: Fpga framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268, 2016.
- [34] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated fpgas: Fpga framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268. IEEE, 2016.

-
- [35] Wei Ding, Zeyu Huang, Zunkai Huang, Li Tian, Hui Wang, and Songlin Feng. Designing efficient accelerator of depthwise separable convolutional neural network on fpga. *Journal of Systems Architecture*, 97:278–286, 2019.
 - [36] Oussama Djedidi and Mohand A. Djeziri. Power profiling and monitoring in embedded systems: A comparative study and a novel methodology based on narx neural networks. *Journal of Systems Architecture*, 111:101805, 2020.
 - [37] Tiziana Fanni, Lin Li, Timo Viitanen, Carlo Sau, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, and Shuvra S. Bhattacharyya. Hardware design methodology using lightweight dataflow and its integration with low power techniques. *Journal of Systems Architecture*, 78:15–29, 2017.
 - [38] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The leap fpga operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.
 - [39] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 47–56, 2012.
 - [40] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.
 - [41] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. Deep learning with int8 optimization on xilinx devices. *White Paper*, 2016.
 - [42] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. Fpdeep: Acceleration and load balancing of cnn training on fpga clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 81–84, 2018.
 - [43] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. Fpdeep: Acceleration and load balancing of cnn training on fpga clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 81–84. IEEE, 2018.
 - [44] Tong Geng, Chunshu Wu, Cheng Tan, Bo Fang, Ang Li, and Martin Herbordt. Cqnn: a cgpa-based qnn framework. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.
 - [45] Paul R Genssler, Oliver Knodel, and Rainer G Spallek. Securing virtualized fpgas for an untrusted cloud. In *Proceedings of the International Conference on Embedded Systems, Cyber-physical Systems, and Applications (ESCS)*, pages 3–9. The Steering Committee of The World Congress in Computer Science, Computer ..., 2018.
 - [46] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.
 - [47] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.
 - [48] Google. cloud, <https://google.com/cloud>.
 - [49] Yijin Guan, Hao Liang, et al. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *25th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2017.
 - [50] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, 2017.

- [51] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2017.
- [52] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.
- [53] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), March 2019.
- [54] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML’15*, page 1737–1746. JMLR.org, 2015.
- [55] Philipp Gysel. Ristretto: Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1605.06402, 2016.
- [56] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5784–5789, 2018.
- [57] Stefan Hadjis and Kunle Olukotun. Tensorflow to cloud fpgas: Tradeoffs for accelerating deep neural networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 360–366, 2019.
- [58] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.
- [59] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.
- [60] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.
- [61] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
- [62] Ke He, Bo Liu, Yu Zhang, Andrew Ling, and Dian Gu. Fecaffe: Fpga-enabled caffe with openc1 for deep learning training and inference on intel stratix 10. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’20*, page 314, New York, NY, USA, 2020. Association for Computing Machinery.
- [63] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 497–514, 2019.
- [64] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *Proceedings of the 28th USENIX Conference on Security Symposium, SEC’19*, page 497–514, USA, 2019. USENIX Association.
- [65] Lien-Chih Hsu, Ching-Te Chiu, Kuan-Ting Lin, Hsing-Huan Chou, and Yen-Yu Pu. Essa: An energy-aware bit-serial streaming deep convolutional neural network accelerator. *Journal of Systems Architecture*, 111:101831, 2020.
- [66] IBM. Cloud,<https://www.ibm.com/cloud/computing>.
- [67] Qaiser Ijaz, El-Bay Bourennane, Ali Kashif Bashir, and Hira Asghar. Revisiting the high-performance reconfigurable computing for future datacenters. *Future Internet*, 12(4), 2020.
- [68] Sakshi Indolia, Anil Kumar Goswami, S.P. Mishra, and Pooja Asopa. Conceptual understanding of convolutional neural network- a deep learning approach. *Procedia Computer Science*, 132:679–688, 2018. International Conference on Computational Intelligence and Data Science.

- [69] Anca Iordache, Guillaume Pierre, Peter Sanders, Jose Gabriel de F. Coutinho, and Mark Stillwell. High performance in the cloud with fpga groups. In *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC '16*, page 1–10, New York, NY, USA, 2016. Association for Computing Machinery.
- [70] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017.
- [71] Weiwen Jiang, Edwin Hsing-Mean Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving super-linear speedup across multi-fpga for real-time DNN inference. *CoRR*, abs/1907.08985, 2019.
- [72] Weiwen Jiang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Lei Yang, Xianzhang Chen, and Jingtong Hu. Heterogeneous fpga-based cost-optimal design for timing-constrained cnns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2542–2554, 2018.
- [73] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, August 2018.
- [74] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, and Mike Daley. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.
- [75] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. OSDI’18, page 107–127, USA, 2018. USENIX Association.
- [76] Doyun Kim, Han Young Yim, Sanghyuck Ha, Changgwun Lee, and Inyup Kang. Convolutional neural network quantization using generalized gamma distribution. *arXiv preprint arXiv:1810.13329*, 2018.
- [77] Jik-Soo Kim, Bui Quang, Seungwoo Rho, Seoyoung Kim, Sangwan Kim, Vincent Breton, and Soonwook Hwang. Towards effective scheduling policies for many-task applications: practice and experience based on htcaas. *Concurrency and Computation: Practice and Experience*, 29(21):e4242, 2017.
- [78] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.
- [79] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. Virtualizing reconfigurable hardware to provide scalability in cloud architectures. In *International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*, 2017.
- [80] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.
- [81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.
- [82] Martin Langhammer and Bogdan Pasca. Floating-point dsp block architecture for fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 117–125, 2015.
- [83] Fengfu Li and Bin Liu. Ternary weight networks. *ArXiv*, abs/1605.04711, 2016.
- [84] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, 2016.
- [85] Xiangwei Li and Douglas L. Maskell. Time-multiplexed fpga overlay architectures: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 24(5), July 2019.
- [86] Xiangwei Li, Kizheppatt Vipin, Douglas L. Maskell, Suhaib A. Fahmy, and Abhishek Kumar Jain. High throughput accelerator interface framework for a linear time-multiplexed fpga overlay. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

- [87] Zhengjie Li, Yufan Zhang, Jian Wang, and Jinmei Lai. A survey of fpga design for ai era. *Journal of Semiconductors*, 41:021402, 02 2020.
- [88] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. *CoRR*, abs/1511.06393, 2015.
- [89] Irene Lopatovska, Katrina Rink, Ian Knight, Kieran Raines, Kevin Cosenza, Harriet Williams, Perachya Sorsche, David Hirsch, Qi Li, and Adrianna Martinez. Talk to me: Exploring user interactions with the amazon alexa. *Journal of Librarianship and Information Science*, 51(4):984–997, 2019.
- [90] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. *arXiv preprint arXiv:1810.01875*, 2018.
- [91] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.
- [92] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.
- [93] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2017.
- [94] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 45–54, New York, NY, USA, 2017. Association for Computing Machinery.
- [95] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2016.
- [96] Yufei Ma, Naveen Suda, Yu Cao, Sarma Vrudhula, and Jae sun Seo. Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration*, 62:14–23, 2018.
- [97] Yufei Ma, Naveen Suda, et al. Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration*, 62:14–23, 2018.
- [98] Kaspar Matas, Tuan La, Nikola Grunchevski, Khoa Pham, and Dirk Koch. Invited tutorial: Fpga hardware security for datacenters and beyond. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery.
- [99] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different - recovering neural network quantization error through weight factorization. *CoRR*, abs/1902.01917, 2019.
- [100] Szymon Migacz. 8-bit inference with tensorrt. In *GPU technology conference*, volume 2, page 7, 2017.
- [101] Sparsh Mittal. A survey of fpga-based accelerators for convolutional neural networks. *Neural computing and applications*, 32(4):1109–1139, 2020.
- [102] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.
- [103] Diksha Moolchandani, Anshul Kumar, and Smruti R. Sarangi. Accelerating cnn inference on asics: A survey. *Journal of Systems Architecture*, 113:101887, 2021.
- [104] Shyam Patidar, Dheeraj Rane, and Pritesh Jain. A survey paper on cloud computing. In *2012 second international conference on advanced computing & communication technologies*, pages 394–398. IEEE, 2012.

- [105] Sasanka Potluri, Alireza Fasih, Laxminand Kishore Vutukuru, Fadi Al Machot, and Kyandoghene Kyamakya. Cnn based high performance computing for real time image processing on gpu. In *Proceedings of the Joint INDS'11 ISTET'11*, pages 1–7, 2011.
- [106] Julia Powles and Hal Hodson. Google deepmind and healthcare in an age of algorithms. *Health and technology*, 7(4):351–367, 2017.
- [107] Sunil Puranik, Mahesh Barve, Dhaval Shah, Sharad Sinha, Rajendra Patrikar, and Swapnil Rodi. Key-value store using high level synthesis flow for securities trading system. In *2020 International Conference on Computing, Electronics Communications Engineering (iCCECE)*, pages 237–242, 2020.
- [108] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.
- [109] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.
- [110] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong 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*, FPGA '16, page 26–35, New York, NY, USA, 2016. Association for Computing Machinery.
- [111] M. Quraishi, E. Tavakoli, and F. Ren. A survey of system architectures and techniques for fpga virtualization. *IEEE Transactions on Parallel and Distributed Systems*, 32(09):2216–2230, sep 2021.
- [112] Atul Rahman, Jongeun Lee, and Kiyoun Choi. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1393–1398, 2016.
- [113] N. Raspa, G. Natale, M. Bacis, and M. D. Santambrogio. A framework with cloud integration for cnn acceleration on fpga devices. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2018.
- [114] Niccolò Raspa, Giuseppe Natale, Marco Bacis, and Marco D Santambrogio. A framework with cloud integration for cnn acceleration on fpga devices. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 170–177. IEEE, 2018.
- [115] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.
- [116] Sahand Salamat, Behnam Khaleghi, Mohsen Imani, and Tajana Rosing. Workload-aware opportunistic energy efficiency in multi-fpga platforms. *CoRR*, abs/1908.06519, 2019.
- [117] Luca B. Saldanha and Christophe Bobda. An embedded system for handwritten digit recognition. *Journal of Systems Architecture*, 61(10):693–699, 2015. Special section on Architecture of Computing Systems edited by Editors: Wolfgang Karl, Erik Maehle, Kay Römer, Eduardo Tovar, Martin Danek Special section on Testing, Prototyping, and Debugging of Multi-Core Architectures edited by Editors: Frank Hannig and Andreas Herkersdorf Special section on Embedded Vision Architectures and Applications edited by Editors: Christophe Bobda, Walter Stechele, Ali Ahmadinia and Miaoqing Huang.
- [118] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.
- [119] Ahmed Shafee and Tasneem A. Awaad. Privacy attacks against deep learning models and their countermeasures. *Journal of Systems Architecture*, 114:101940, 2021.
- [120] Junnan Shan et al. Power-optimal mapping of cnn applications to cloud-based multi-fpga platforms. *IEEE Transactions on Circuits and Systems*, 2020.

- [121] Junnan Shan, Mihai T. Lazarescu, Jordi Cortadella, Luciano Lavagno, and Mario R. Casu. Cnn-on-aws: Efficient allocation of multikernel applications on multi-fpga platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(2):301–314, 2021.
- [122] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.
- [123] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.
- [124] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.
- [125] Junzhong Shen, Deguang Wang, You Huang, Mei Wen, and Chunyuan Zhang. Scale-out acceleration for 3d cnn-based lung nodule segmentation on a multi-fpga system. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.
- [126] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [127] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [128] Rym Skhiri, Virginie Fresse, et al. From fpga to support cloud to cloud of fpga: State of the art. *International Journal of Reconfigurable Computing*, 2019.
- [129] Hayden Kwok-Hay So and Cheng Liu. *FPGA Overlays*, pages 285–305. Springer International Publishing, Cham, 2016.
- [130] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2017.
- [131] Rastislav J.R. Struharik, Bogdan Z. Vukobratović, Andrea M. Erdeljan, and Damjan M. Rakanović. Conna—hardware accelerator for compressed convolutional neural networks. *Microprocessors and Microsystems*, 73:102991, 2020.
- [132] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.
- [133] Tencent. Tencent cloud: Instance type fpga fx2, <https://intl.cloud.tencent.com/document/product/213/11518fx2>.
- [134] Hsin-Yu Ting, Tootiya Giyahchi, Ardalan Amiri Sani, and Eli Bozorgzadeh. Dynamic sharing in multi-accelerators of neural networks on an fpga edge device. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 197–204, 2020.
- [135] Stephen Tridgell, Martin Kumm, Martin Hardieck, David Boland, Duncan Moss, Peter Zipf, and Philip H. W. Leong. Unrolling ternary neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, 12(4), October 2019.
- [136] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees 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*, FPGA '17, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery.
- [137] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A survey on fpga virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317, 2018.
- [138] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource elastic virtualization for fpgas using opencl. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117, 2018.

- [139] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.
- [140] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. IEEE, 2016.
- [141] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.
- [142] Stylianos I. Venieris and Christos-Savvas Bouganis. f-cnnx: A toolflow for mapping multiple convolutional neural networks on fpgas. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 381–3817, 2018.
- [143] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Comput. Surv.*, 51(3), June 2018.
- [144] Stylianos I Venieris, Ioannis Panopoulos, Ilias Leontiadis, and Iakovos S Venieris. How to reach real-time ai on consumer devices? solutions for programmable and custom architectures. *arXiv preprint arXiv:2106.15021*, 2021.
- [145] Kizheppatt Vipin and Suhaib A. Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4), July 2018.
- [146] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. Lutnet: Learning fpga configurations for highly efficient neural network inference. *IEEE Transactions on Computers*, 69(12):1795–1808, 2020.
- [147] Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, Michael Lang, and Ioan Raicu. Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. *Concurrency and Computation: Practice and Experience*, 28(1):70–94, 2016.
- [148] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling fpgas in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086, 2015.
- [149] Qingcheng Xiao and Yun Liang. Fune: An fpga tuning framework for cnn acceleration. *IEEE Design Test*, 37(1):46–55, 2020.
- [150] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.
- [151] Xilinx. Axi4 dma, vivado design suite document (pg021).
- [152] Xilinx. support documentation (ug761).
- [153] Xilinx®. Vivado design suite user guide: High-level synthesis (ug902).
- [154] Xilinx. Xilinx axi reference guide (ug762).
- [155] Xilinx. Xilinx zynq-7000 series, <https://www.xilinx.com/products/silicon-devices/soc.html>.
- [156] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yi Shan, and Yu Wang. Dnnvm: End-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2668–2681, 2019.
- [157] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. A parallel bandit-based approach for autotuning fpga compilation. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17*, page 157–166, New York, NY, USA, 2017. Association for Computing Machinery.

- [158] Zhe Xu and Ray C.C. Cheung. Binary convolutional neural network acceleration framework for rapid system prototyping. *Journal of Systems Architecture*, 109:101762, 2020.
- [159] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9(4):611–629, 2018.
- [160] Fan Yao, A. S. Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. *ArXiv*, abs/2003.13746, 2020.
- [161] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1463–1480, 2020.
- [162] Sadeqh Yazdanshenas and Vaughn Betz. The costs of confidentiality in virtualized fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(10):2272–2283, 2019.
- [163] Sadeqh Yazdanshenas and Vaughn Betz. The costs of confidentiality in virtualized fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(10):2272–2283, 2019.
- [164] Xiaoyu Yu, Yuwei Wang, Jie Miao, Ephrem Wu, Heng Zhang, Yu Meng, Bo Zhang, Biao Min, Dewei Chen, and Jianlin Gao. A data-center fpga acceleration platform for convolutional neural networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 151–158. IEEE, 2019.
- [165] Jure Zbontar, Yann LeCun, et al. Stereo matching by training a convolutional neural network to compare image patches. *J. Mach. Learn. Res.*, 17(1):2287–2318, 2016.
- [166] Hanqing Zeng, Ren Chen, Chi Zhang, and Viktor Prasanna. A framework for generating high throughput cnn implementations on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18*, page 117–126, New York, NY, USA, 2018. Association for Computing Machinery.
- [167] Shulin Zeng, Guohao Dai, Hanbo Sun, Kai Zhong, Guangjun Ge, Kaiyuan Guo, Yu Wang, and Huazhong Yang. Enabling efficient and flexible fpga virtualization for deep learning in the cloud. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 102–110. IEEE, 2020.
- [168] Shulin Zeng, Guohao Dai, Kai Zhong, Hanbo Sun, Guangjun Ge, Kaiyuan Guo, Yu Wang, and Huazhong Yang. Enable efficient and flexible fpga virtualization for deep learning in the cloud. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '20*, page 317, New York, NY, USA, 2020. Association for Computing Machinery.
- [169] Yue Zha and Jing Li. Virtualizing fpgas in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 845–858, New York, NY, USA, 2020. Association for Computing Machinery.
- [170] Yue Zha and Jing Li. Virtualizing fpgas in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.
- [171] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15*, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.
- [172] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.
- [173] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2018.
- [174] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2019.

-
- [175] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design, ISLPED '16*, page 326–331, New York, NY, USA, 2016. Association for Computing Machinery.
 - [176] Min Zhang, Linpeng Li, Hai Wang, Yan Liu, Hongbo Qin, and Wei Zhao. Optimized compression for implementing convolutional neural networks on fpga. *Electronics*, 8(3), 2019.
 - [177] W. Zhang, J. Zhang, M. Shen, G. Luo, and N. Xiao. An efficient mapping approach to large-scale dnns on multi-fpga architectures. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1241–1244, 2019.
 - [178] Wentai Zhang, Jiaxi Zhang, Minghua Shen, Guojie Luo, and Nong Xiao. An efficient mapping approach to large-scale dnns on multi-fpga architectures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1241–1244. IEEE, 2019.
 - [179] Xiaofan Zhang, Xinheng Liu, Anand Ramachandran, Chuanhao Zhuge, Shibin Tang, Peng Ouyang, Zuofu Cheng, Kyle Rupnow, and Deming Chen. High-performance video content recognition with long-term recurrent convolutional network for fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.
 - [180] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.
 - [181] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.
 - [182] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.
 - [183] Yongmei Zhou and Jingfei Jiang. An fpga-based accelerator implementation for deep convolutional neural networks. In *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, volume 01, pages 829–832, 2015.

Conception d'une plate-forme multi-FPGA dans le cloud pour les applications de réseaux de neurones

RÉSUMÉ

L'évolution rapide des réseaux de neurones a conduit à des architectures de réseaux nécessitant des capacités de calculs importantes avec des structures de réseau de plus en plus profondes. Le déploiement des réseaux de neurones sur des composants CPU/GPU fait face aux défis de la consommation énergétique. Pour faire face à ce problème de consommation énergétique, l'utilisation de circuits reconfigurables, comme les FPGA (Field Programmable Gate Array) est devenu une alternative de plus en plus envisagée. Cependant, le déploiement de réseaux sur des FPGA nécessite des outils de conception matérielle spécifiques et une solide expertise matérielle pour mener à bien leur conception jusqu'à l'implémentation finale sur FPGA.

L'objectif de la thèse est de fournir une infrastructure de cloud computing basée sur des FPGAs dédiés aux ingénieurs en Machine Learning pour exécuter différents modèles de CNN (Convolutional Neural Network) sur diverses plateformes FPGA sans connaissance matérielle. L'infrastructure offre plusieurs IPs de CNN matériels, qui ont soit une structure haut débit via des mise en œuvre de pipeline, soit optimisent les ressources matérielles, ou ont une structure apportant un compromis entre les ressources et le temps de calcul. Ces IP sont conçus pour générer différentes architectures matérielles de CNN sur des FPGAs en fonction des exigences des ingénieurs en Machine Learning. L'infrastructure intègre également des modèles mathématiques, qui estiment les ressources nécessaires des IPs en fonction de leurs paramètres et sans passer par l'étape de synthèse (qui est très coûteuse en temps). Cette estimation peut aider à allouer aux mieux les ressources FPGAs dans le cloud et choisir le ou les FPGAs appropriés. Enfin, un outil de quantification est conçu pour compresser la taille du réseau en diminuant la taille des données des CNN sur FPGA.

Afin de compléter la fonctionnalité de l'infrastructure, plusieurs cas d'utilisation sont également développés pour couvrir tous les cas d'usages des applications associées aux réseaux de neurones. Cette thèse présente également le cycle de vie de cette infrastructure afin de mener une analyse approfondie du fonctionnement de l'infrastructure pour différents ingénieurs en Machine Learning dans divers cas d'utilisation. L'infrastructure proposée peut analyser les besoins des utilisateurs pour d'autres cas d'utilisation que l'inférence, déployer l'architecture matérielle du CNN sur le FPGA approprié et mettre en œuvre des techniques d'optimisations si nécessaire.

Mots clés: Réseau Neuron Convolutif, FPGA, Accélérateur, Informatique en nuage.

Design of multi-FPGAs platform in the cloud for neural network applications

ABSTRACT

The rapid innovation of neural network algorithms has led to neural network architectures with more calculations and deeper structures. However, the neural network deployment on traditional devices such as CPU/GPU faces energy consumption challenges. In this case, Field Programmable Gate Array (FPGA) has become an alternative to realizing neural networks because of its efficient energy and reconfigurability. However, the deployment of neural network engineers on FPGAs requires specific hardware design tools and solid hardware knowledge to complete the design to the final implementation.

The objective of the thesis is to provide an FPGA-based Cloud computing infrastructure dedicated to machine learning engineers to perform different CNN models on various FPGA platforms without hardware acknowledges. The infrastructure offers multiple CNN hardware IPs, which have a high-throughput structure through pipelines, or save hardware resource consumption, or have a structure that strikes a balance between the two. These IPs are designed to generate different CNN hardware architectures on FPGAs according to the requirements of machine learning engineers. The infrastructure also involves several mathematical models, which estimate the resource usage of the two IPs developed. This estimation can help allocate FPGA resources well in the cloud. Finally, a quantization tool is designed to compress the network size with any bit width for the implementation on the FPGA.

In order to complete the functionality of the infrastructure, several use cases are also developed to achieve the multiple usages of the neural network applications. This thesis also provides an overview of the life cycle for this infrastructure to conduct an in-deep analysis of how the infrastructure works for different machine learning engineers in various use cases. The proposed infrastructure can analyze the user needs of other use cases, deploy the CNN hardware architecture on the appropriate FPGA, and implement optimization techniques when necessary.

Key words: *Convolution Neural Network, FPGA, Accelerator, Cloud computing.*