DESPLIEGUE DE REDES NEURONALES MEDIANTE PYNQ
DEPLOYMENT OF NEURAL NETWORKS THROUGH PYNQ

TRABAJO FIN DE MÁSTER
CURSO 2022-2023

BY
WENBO SUN

DIRECTOR
ALBERTO A. DEL BARRIO

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

NOTA: 5

SEPTIEMBRE 2023

THANKS

I am very grateful to my teacher Alberto Del Barrio. From the topic selection of the thesis, data collection to the completion of the thesis writing, I received enthusiastic help from the teacher.

First of all, I would like to thank my mentor Alberto A. del Barrio. He put forward a lot of valuable opinions on my research, which gave me a goal and direction for my thesis writing.

Secondly, I would like to thank all the teachers who taught me, they gave me knowledge and good teaching.

Finally, I would like to thank the classmates who accompanied me during the study, have a good spirit of cooperation, and often brainstorm together. Thank you for all the helpful suggestions and comments they gave me.

ABSTRACT

DEPLOYMENT OF NEURAL NETWORKS THROUGH PYNQ

The PYNQ platform provides a python interface for accessing FPGA resources, which gives us the opportunity to efficiently deploy neural network models on FPGA to achieve high-performance and real-time image classification and target detection tasks. This hardware-accelerated approach can provide faster inference speed and lower power consumption than software-accelerated approach.

In this research and development project, our main research objective is to deploy neural networks on PYNQ. I have used the PYNQ-Z1 development board for experiments. Four type of networks have been deployed, namely: a YOLO network, a BNN network, a ResNet network and a MobileNetv2 network. After deployment, I have compared their accuracy and measured their execution time on hardware, achieving promising results for a resource-constrained device as the Z1 board.

Keywords

PYNQ, FPGA, Computer vision, neural network, object detection,

deep learning, machine learning

INDEX OF CONTENTS

Chapter1 Introduction 6

Section 1.1 - Motivation 6

Section 1.2 - Objective 7

Chapter2 State of the art 8

Section 2.1 - Convolutional Neural Network, CNNs 8

Section 2.2 - Binary Neural Networks, BNNs 16

Section 2.3 - YOLO 18

Section 2.4 - ResNet 19

Section 2.5 - MobileNetV2 20

Chapter3 Data Sets 22

Section 3.1 - CIFAR-10 22

Section 3.2 - COCO 23

Chapter4 Hardware and Software 25

Section 4.1 - PYNQ-Z1 25

Section 4.2 - Vivado 27

Section 4.3 - Docker 28

Section 4.4 - PuTTY 28

Chapter5 Frameworks 29

Section 5.1 - PyTorch 29

Section 5.2 - BREVITAS 29

Section 5.3 - TENSORFLOW 30

Section 5.4 - FINN 31

Section 5.4.1 - FINN-CNV 34

Section 5.5 - Tensil-AI 37

Chapter6 Methodology and Experiments 37

Section 6.1 - Methodology 37

Section 6.2 - Synthesis results 39

Section 6.2.1 - COCO dataset 40

Section 6.2.1.1 - YOLOv2 model 40

1) FPGA-Based Optimization 40

2) HLS synthesis 41

3) Block Design..42

Section 6.2.2 - CIFAR-10 dataset 44

Section 6.2.2.1 - XNORNet 44

Section 6.2.2.2 - ResNet 47

Section 6.2.2.3 - MobileNetv2 49

Section 6.3 - Execution time 52

Section 6.3.1 - Time for each network to run on PYNQ using the FPGA accelerator: 52

Section 6.4 - Accuracy 54

Section 6.4.1 - Top-1 Accuracy for each network to run on PYNQ using the FPGA accelerator: 54

Chapter8 Conclusions and future work..55

Section 8.1 - Conclusions 55

Section 8.2 - Future Work 56

REFERENCES 58

Chapter1 Introduction

In recent years, there has been significant interest in the field of neural networks [1]. However, neural networks face challenges in terms of computational requirements and real-time performance. These challenges have prompted researchers to explore alternative solutions. Field Programmable Gate Arrays (FPGAs) offer flexibility and configurability, making them well-suited for high parallel computing tasks. PYNQ, a Python-based development platform, provides an excellent environment for leveraging the power of FPGAs.

The deployment and acceleration of neural networks on the PYNQ platform have garnered substantial attention in recent years. Researchers have focused not only on improving performance and speed but also on model compression and optimization techniques. Consequently, I will embark on the task of deploying mainstream neural network models on the PYNQ platform.

Section 1.1 - Motivation

The motivation behind this research stems from the need for enhanced hardware acceleration, real-time performance, and energy efficiency in the deployment of neural network models. The PYNQ platform, which combines Python programming capabilities with the Xilinx Zynq platform, presents an opportunity to address these requirements effectively.

In terms of hardware acceleration, the integration of programmable logic (FPGA) and embedded processors in the ZYNQ platform enables the deployment of neural

network models on FPGAs. Leveraging the power of FPGAs allows for hardware acceleration, leading to increased inference speed and efficiency.

Real-time performance is critical in various application scenarios such as real-time video analysis, robot control, and unmanned driving. The FPGA on the PYNQ platform provides low-latency hardware computing capabilities, enabling neural network models to complete inference tasks within tight time constraints, thus meeting real-time performance requirements.

Furthermore, considering power constraints in embedded systems and mobile devices, reducing energy consumption is paramount. By deploying neural network models on PYNQ and harnessing the hardware acceleration capabilities of FPGAs, high-performance inference can be achieved while minimizing power consumption. Compared to traditional general-purpose processors, FPGAs excel at highly parallel computing tasks, resulting in lower power consumption.

From a software perspective, FPGAs have traditionally been programmed in RTL languages such as VHDL/Verilog, and even C in the past decade. However, using Python provides programmers with many functions, and the interface can be called to realize many functions on PYNQ.

Section 1.2 - Objective

The objective of this project is to test the capacity of the PYNQ platform by deploying several DNNs on a Z1 board and compare it with PYNQ's own arm core and ordinary CPUs.

In this project, the primary focus will be on deploying various mainstream neural network models on the PYNQ platform. These models will be carefully selected and tested using widely recognized datasets. The deployment process will involve model compression and optimization techniques to ensure efficient utilization of the PYNQ platform's capabilities.

The distinctive features of the PYNQ platform make it an ideal choice for this project. Once the deployment phase is completed, an extensive evaluation will be conducted to assess the hardware acceleration capabilities achieved. The aim is to demonstrate notable improvements in reasoning speed and efficiency by successfully deploying a range of mainstream models on the PYNQ platform.

Chapter2 State of the art

Section 2.1 - Convolutional Neural Network, CNNs

Convolutional Neural Networks (CNNs) [2] are a deep learning model widely used in computer vision and image processing. CNNs are specifically designed to process data with a grid structure, such as images or videos [10].

By using components such as convolutional layers, pooling layers, and fully connected layers, it can effectively extract features from images and implement tasks such as image classification, object detection, and image segmentation. CNNs use the

characteristics of local connection, weight sharing and pooling to make the network invariant to the translation, rotation and scale change of the image.

As the application of CNNs in computer vision tasks, CNV(Convolutional Neural Network for Vision)[20] usually uses a specific network structure and layer configuration to meet the needs of image processing. CNV models may be designed and optimized for different vision tasks, such as image classification, object detection, or image segmentation.

A CNNs neural network model has more than classic 3-layer structure, but the classic is the following three layers:

1. Convolutional Layers
2. Pooling Layers
3. Fully Connected Layers

When we use this model, we usually add “Activation Functions”. Using activation functions enables CNNs to learn and represent more complex functions.

Let's briefly describe:

The convolutional layer is responsible for extracting local features in the image.

- Pseudocode for convolutional layers:

```
for(r=0;r<R;r++)
 for(c=0;c<C;c++)
  for(m=0;m<M;m++)
   for(n=0;n<N;n++){
    for(ky=0;ky<K;ky++)
     for(kx=0;kx<K;kx++){
          pixelL(m,r,c)+=pixelL-1(n,r*S+ky,c*S+kx)*weightL-1(m,n,ky,kx);}
     pixelL(m,r,c)=pixelL(m,r,c)+bias(m);}
```

- for(r=0;r<R;r++) and for(c=0;c<C;c++): These loops iterate over the rows and columns of the output feature map. Here, R represents the number of rows in the output feature map, and C represents the number of columns. These loops iterate through every position in the output feature map.
- for(m=0;m<M;m++): This loop iterates over the channels or depth of the output feature map. M represents the number of output channels, where each channel corresponds to a different feature.
- for(n=0;n<N;n++): This loop iterates over the channels or depth of the input feature map. N represents the number of input channels, where each channel corresponds to a different input feature.
- for(ky=0;ky<K;ky++) and for(kx=0;kx<K;kx++): These nested loops are used to iterate over the height and width of the convolutional kernel. K represents the size of the kernel, which is typically square.

Inside these nested loops, the following operations occur:

- pixelL(m,r,c): Represents the pixel value at position (r, c) of the m-th channel of the output feature map.
- pixelL-1(n,r*S+ky,c*S+kx): Represents the pixel value at position (r*S+ky, c*S+kx) of the n-th channel of the input feature map, where S is the stride.
- weightL-1(m,n,ky,kx): Represents the weight of the convolutional kernel used for element-wise multiplication with the input feature.
- bias(m): Represents the bias term added to the result for the m-th channel of the output feature map.

Inside the innermost loops, the code iterates over each element of the convolutional kernel, performing element-wise multiplication with the input feature and accumulating the results. This calculates the pixel value at position (r, c) of the m-th channel of the output feature map. Finally, the bias term is added to obtain the final output pixel value.

The pooling layer is used to greatly reduce the parameter magnitude (dimension

reduce).

- Pseudocode for pooling layer:

```
for (r=0;r<R;r++)
 for (c=0;c<C;c++)
  for (m=0;m<M;m++){
    pixel_L(m,r,c) = Max(pixel_{L-1}(n,r×S+ky,c×S+kx));
    ky,kx∈{0,1,···,K-1} }
```

- for (r=0;r<R;r++) and for (c=0;c<C;c++): These loops iterate over the rows and columns of the output feature map. Here, R represents the number of rows in the output feature map, and C represents the number of columns. These loops iterate through every position in the output feature map.
- for (m=0;m<M;m++): This loop iterates over the channels or depth of the output feature map. M represents the number of output channels, where each channel corresponds to a different feature.

Inside the loops, the following operation occurs:

- pixelL(m,r,c) = Max(pixelL-1(n,r×S+ky,c×S+kx)): This line calculates the value of the m-th channel at position (r, c) in the output feature map. It does so by taking the maximum value from a neighborhood of pixels in the corresponding n-th channel of the input feature map.
  - ky and kx iterate through the neighborhood of pixels within a window of size K×K. This window slides over the input feature map with a step size defined by S.
  - pixelL-1(n,r×S+ky,c×S+kx) represents the pixel value at position (r×S+ky, c×S+kx) in the n-th channel of the input feature map.
  - Max is used to calculate the maximum value within the specified neighborhood. This operation is commonly known as max pooling.

The pooling layer effectively reduces the spatial dimensions of the feature map (width and height) while retaining the most prominent features in each local

neighborhood defined by the K×K window. This reduces the number of parameters and computations in the network, aiding in dimension reduction and feature selection, which can improve the network's efficiency and generalization.

The fully connected layer is similar to the part of a traditional neural network to output the desired result.

The role of the activation function is to introduce nonlinear properties, enabling CNNs to learn and represent more complex functions. Commonly used activation functions include ReLU (Rectified Linear Unit), Sigmoid, and Tanh. ReLU is the most commonly used activation function, which can effectively solve the problem of gradient disappearance and accelerate the training process of the network [12][13]. Common Activation Functions: The passage mentions several common activation functions, including ReLU, Sigmoid, and Tanh.

ReLU (Rectified Linear Unit): ReLU is one of the most widely used activation functions. It is defined as $f(x) = max(0, x)$, meaning that it outputs zero for input values less than zero and retains the input for values greater than zero. ReLU is favored for its simplicity and its effectiveness in addressing the vanishing gradient problem in many cases. Its nonlinear nature enables neural networks to learn complex patterns and features

Sigmoid: The Sigmoid function maps inputs to values between 0 and 1, characterized by a smooth S-shaped curve. It is commonly used in binary classification problems but may encounter the vanishing gradient problem in deep networks.

Tanh (Hyperbolic Tangent): Tanh maps inputs to values between -1 and 1, also exhibiting an S-shaped curve. Unlike Sigmoid, Tanh has a zero-centered output, making it more suitable for neural networks. However, similar to Sigmoid, it can still face the vanishing gradient problem.

1. Convolution-extracting features

The operation process of the convolution layer is shown in the Figure[1] below, and the complete picture is scanned with the convolution kernel:

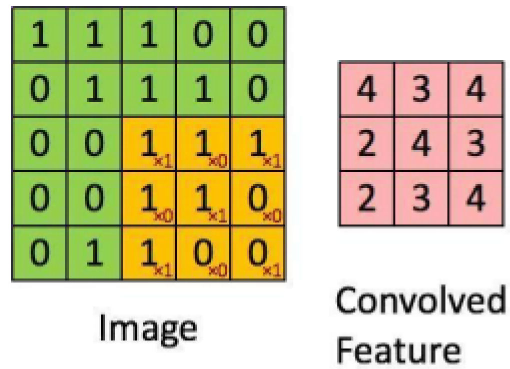


Figure 1. Convolutional layer extraction features.

The convolutional layer is a key building block in CNNs. It operates by convolving an input image with a convolutional kernel (also known as a filter). The convolutional kernel is a small matrix that slides over the input image, element-wise multiplying and summing the local regions of the input to generate an output feature map. During the convolution operation, the kernel slides over the input image with a certain stride, starting from the top-left corner and moving row by row and column by column, covering different positions across the entire image. At each position, the convolutional kernel performs the convolution operation with the local region of the input image, generating the corresponding value in the output feature map.

2. Pooling layers (down sampling) - reduce the dimensionality of feature maps, thereby reducing computational complexity and enhancing model robustness.

A pooling layer is a simple down sampling operation that can greatly reduce the dimensionality of the data. The process is as follows:

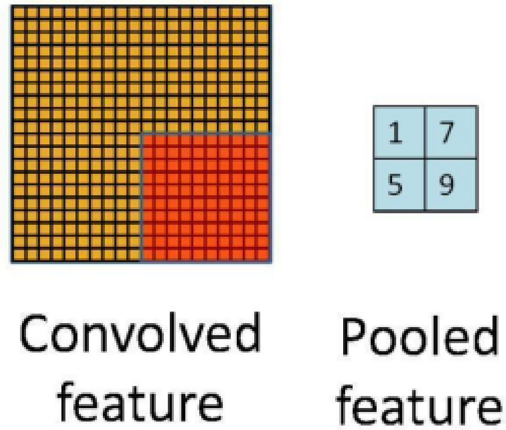


Figure 2. Pooling layer data dimensionality reduction.

Pooling operations are typically applied after convolutional layers. They can be executed on each channel (or depth) of the feature maps.

The pooling operation divides the feature map into non-overlapping regions, usually rectangular in shape. Values within each region are aggregated into a single value. This aggregation process typically falls into one of two common types:

Max Pooling: Selects the maximum value within each region as the aggregation result.

Average Pooling: Computes the average of values within each region as the aggregation result.

The aggregated results form a new feature map with reduced spatial dimensions while preserving essential information from the original feature map.

Pooling operations often have a stride parameter, which controls how the pooling window moves across the feature map. This stride affects the size of the output feature map.

3. Fully connected layer - generate output

The fully connected layer is the last step of CNNs, which will input the data processed by the convolution layer and pooling layer, and generate the final desired output result. The role of the fully connected layer is to perform operations on the data after dimensionality reduction in order to obtain the desired output. If it is not processed by the convolution layer and the pooling layer, the amount of data will be very large, the calculation cost will be high, and the efficiency will be low. Therefore, the existence of the fully connected layer can improve the calculation efficiency and produce accurate output results.

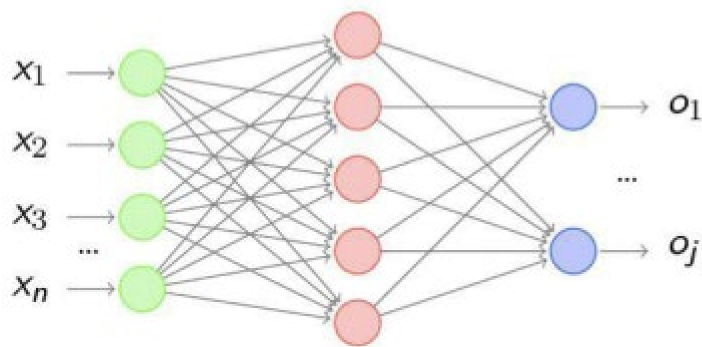


Figure 3. Fully connected layer output result.

Section 2.2 - Binary Neural Networks, BNNs

BNNs (Binary Neural Networks) [4] is a special type of neural network in which the weights of neurons and the value of the activation function take only two values: +1 and -1, that is, in binary form. Compared with traditional floating-point weights, BNNs use of binary weights can greatly reduce the storage requirements and computational complexity of the model.

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise,} \end{cases} \qquad x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x), \\ -1 & \text{with probability } 1 - p, \end{cases}$$

Figure 4. Binarization – Deterministic and Stochastic.

We use BNNs on FPGAs mainly for several reasons:

1. Computational efficiency:

   BNNs use binary weights and activation functions, which can greatly reduce computational complexity. Implementing BNNs on FPGAs can take advantage of hardware parallelism and high customizability to perform binary computation operations with higher efficiency. The programmability of FPGAs makes it possible to optimize the hardware design to support the parallelism of binary computations, enabling high-speed neural network inference.

2. Storage efficiency:

   BNNs uses binary weights and activation functions to greatly reduce the storage requirements of the model. Compared with traditional neural network

models that use floating-point weights and activation functions, BNNs can save a lot of storage space. This is especially important on resource-constrained FPGA devices that can accommodate larger network models or more network layers.

3. Low power consumption:

   Since the binary calculation operation of BNNs is relatively simple, implementing BNNs on FPGA can reduce power consumption. Binary calculations typically involve fewer operands and fewer memory accesses, reducing power requirements. This is beneficial for power sensitive applications such as mobile devices and embedded systems.

4. Real-time requirements:

   BNNs' efficient reasoning ability on FPGA makes it especially suitable for applications with high real-time requirements. For example, tasks such as object detection, video processing, and autonomous driving often require processing large amounts of data in a short amount of time. By implementing BNNs on FPGAs, it is easier to meet the real-time requirements.

Although BNNs has the above advantages, the accuracy of BNNs on some complex tasks may be relatively low due to the information loss that may be introduced by the binarization of weights and activation values [14]. But our project is to successfully deploy neural networks on PYNQ, so we don't discuss the relationship of accuracy, of course, we also need to ensure a certain degree of accuracy. Here we use XNORNet [19] to implement BNNs network.

Section 2.3 - YOLO

YOLO (You Only Look Once) is a real-time target detection algorithm based on deep learning. Compared with traditional object detection algorithms, YOLO has faster inference speed while maintaining high accuracy.

Now the latest YOLO version is Yolov7 [15], but because our FPGA device is PYNQ-Z1. PYNQ-Z1 is a relatively basic development board with limited processor and memory resources. The network structure and algorithm of yolov4 are more complicated, and it is unlikely to be successfully deployed on PYNQ-Z1. The version of Yolov2 is relatively lightweight, and it is more suitable for implementing target detection on PYNQ-Z1 with limited resources.

YOLOv2 [3] uses the Darknet-19 network structure as its foundation. Darknet-19 is a 19-layer convolutional neural network consisting of convolutional layers, pooling layers, and fully connected layers.

In order to better detect objects of different scales, YOLOv2 introduces multi-scale feature representation. It performs object detection on feature maps at different levels, enabling the algorithm to more accurately locate and classify objects of different sizes.

When YOLOv2 detects objects on lower-resolution feature maps, it introduces fine-grained features to improve the detection accuracy of small objects. This can better capture the details and features of the target.

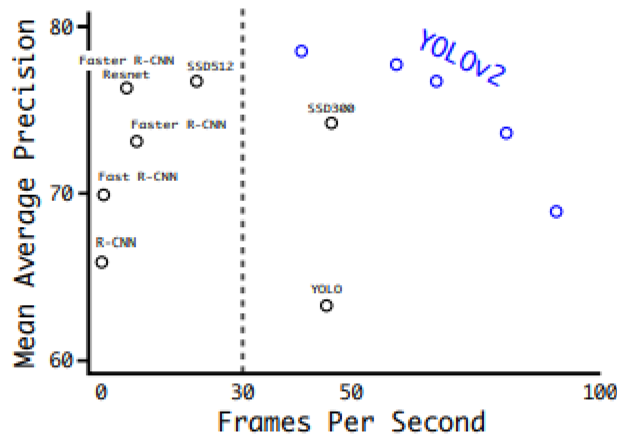


Figure 5. Accuracy and speed on VOC 2007[27].

Section 2.4 - ResNet

ResNet (Residual Networks) [17] is a deep convolutional neural network structure proposed by researchers at the University of Helsinki and Microsoft Research in 2015. The innovation of ResNet is the introduction of the concept of residual learning to solve the problem of gradient disappearance and gradient explosion in deep networks, thus allowing the training of deeper neural networks.

ResNet builds a deep network by introducing a "residual block". Each residual block consists of one or more convolutional layers, and a "skip connection" or "shortcut connection" for skipping a part of the input. Skip connections allow the original input to be passed directly to subsequent layers instead of only through the transformation of the convolutional layer, which ensures that information is passed without loss during training.

The main ResNet variants include ResNet-20, ResNet-34, ResNet-50, ResNet-101, and ResNet-152, etc., and the numbers represent the depth of the layer. Among them, ResNet-50 and deeper models have achieved significant performance improvements in many computer vision tasks, such as image classification, object detection and semantic segmentation.

The advantages of ResNet include:

1. Overcoming the vanishing gradient problem: By skipping connections, ResNet enables gradients to propagate back to earlier layers more easily, avoiding the vanishing gradient problem in deep networks.

2. Training Deeper Networks: ResNet's residual blocks allow training deeper neural networks, which improves model performance on some tasks.

3. Avoid the degradation problem: In theory, a deeper network should perform better on the training set, but in practice, training a deeper network may lead to performance degradation. ResNet helps avoid this with skip connections.

Section 2.5 - MobileNetV2

MobileNetV2 [18] is a lightweight convolutional neural network architecture designed to provide good performance while keeping computation and parameter counts low. It is the successor of MobileNetV1 proposed by Google in 2018, optimized for resource-constrained scenarios such as mobile devices and embedded systems.

The main innovations of MobileNetV2 include:

1. Inverted Residuals: MobileNetV2 introduces an inverted residual structure, which is different from traditional residual blocks. This structure applies depthwise convolution to increase the number of channels when the input dimension is low, and then uses pointwise convolution to reduce the number of channels. Such a design preserves feature expressiveness to a certain extent while reducing computational cost.

2. Linear Bottlenecks: MobileNetV2 uses a linear bottleneck structure to effectively utilize the capacity of the network by matching the number of input channels with the number of output channels. This reduces the dimensionality of the feature map, thereby reducing the amount of computation.

3. Lightweight Feature Integration: MobileNetV2 adopts a feature fusion strategy to fuse feature maps of different resolutions together to improve the expressive ability of multi-scale features.

The improvement of MobileNetV2 over MobileNetV1 makes it achieve a better balance between speed and accuracy. It is suitable for application scenarios with limited computing resources such as image classification, object detection, and semantic segmentation. The design concept of MobileNetV2 provides an efficient selection of neural network models for mobile devices and embedded systems, which can achieve satisfactory performance with limited resources.

Chapter3 Data Sets

To complete this project, we need to use the correct dataset. We choose CIFAR-10 and COCO datasets. The CIFAR-10 dataset is a relatively small-scale dataset that is ideal for rapid verification and prototype testing. It is widely used in image classification tasks. At the same time, despite its small size, CIFAR-10 contains multiple categories, some of which have low discrimination between categories, so it still poses certain challenges to the model. This makes CIFAR-10 a useful tool for assessing a model's ability to generalize. The COCO dataset is large-scale and diverse. This makes COCO ideal for evaluating complex image tasks.

Section 3.1 - CIFAR-10

CIFAR-10 [7] is a widely used image classification dataset created by the Canadian Institute for Advanced Research. The dataset contains 10 different categories of color images, each category contains about 6,000 images, a total of 60,000 images. Each image has a resolution of 32x32 pixels.

The images in the CIFAR-10 dataset are divided into the following 10 categories:

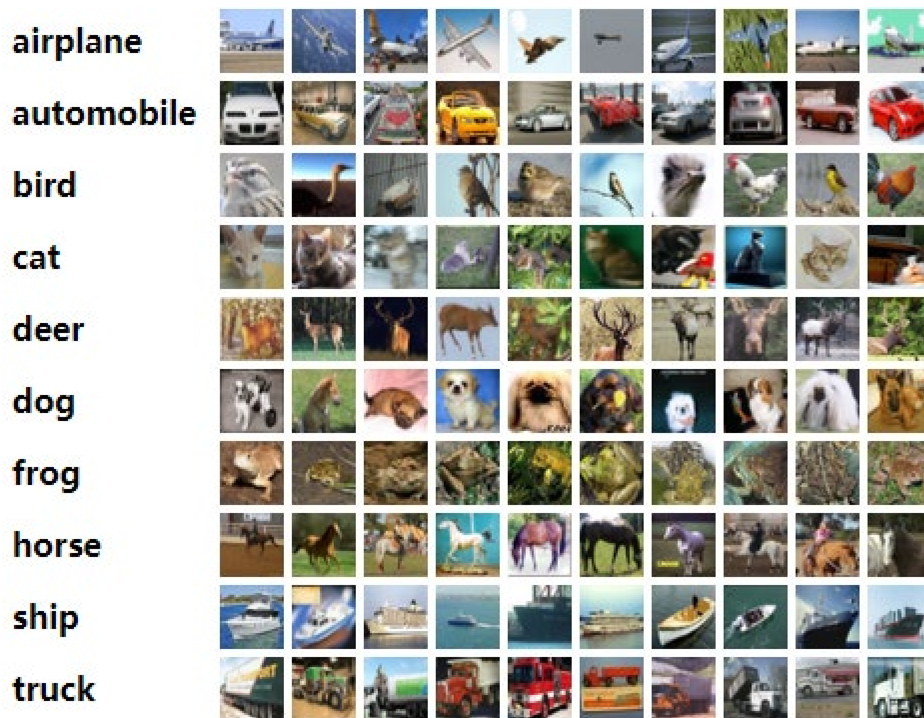


Figure 6. CIFAR-10 categories/outputs.

The CIFAR-10 dataset is widely used in the evaluation and testing of image classification and machine learning algorithms. Due to the small size and rich categories of images, researchers can train and test various image classification models relatively quickly. It is also widely used in the teaching and academic research of deep learning as one of the benchmark datasets for verifying model performance and algorithm effects.

Section 3.2 - COCO

COCO (Common Objects in Context) [8] is a widely used image dataset for the research and evaluation of computer vision tasks such as object detection, image

segmentation, and scene understanding. The COCO dataset was created by Microsoft Research and contains a large number of images with rich scenes and complex objects.

Figure 7. COCO-DATA.

The characteristics of the COCO dataset are as follows:

1. Large scale: The COCO dataset contains more than 200,000 images covering 80 common categories of objects, such as people, animals, vehicles, furniture, etc. Each image is annotated with location and category information of multiple objects.

2. Diversity: The images in the COCO dataset cover a variety of different scenes, including indoor, outdoor, urban, rural, etc. This makes the COCO dataset better adaptable to different real-world scenarios during model training and evaluation.

3. Complexity: Objects in the COCO dataset often have complex shapes, poses, and occlusions. This makes it necessary to consider the diversity and complexity of objects when using the COCO dataset for object detection and image segmentation tasks.

The COCO dataset is widely used in computer vision research and evaluation of algorithms. It provides a rich and diverse dataset that can be used to train and test the performance of object detection and image segmentation algorithms.

Chapter4 Hardware and Software

Section 4.1 - PYNQ-Z1

The PYNQ-Z1[1] board[2] is designed to be used with PYNQ, a new open-source framework that enables embedded programmers to exploit the capabilities of Xilinx Zynq All Programmable SoCs (APSoCs) without having to design programmable logic circuits. Instead the APSoC is programmed using Python, with the code developed and tested directly on the PYNQ-Z1. The programmable logic circuits are imported as hardware libraries and programmed through their APIs in essentially the same way that the software libraries are imported and programmed.

[1] https://digilent.com/shop/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/

[2] http://www.pynq.io/

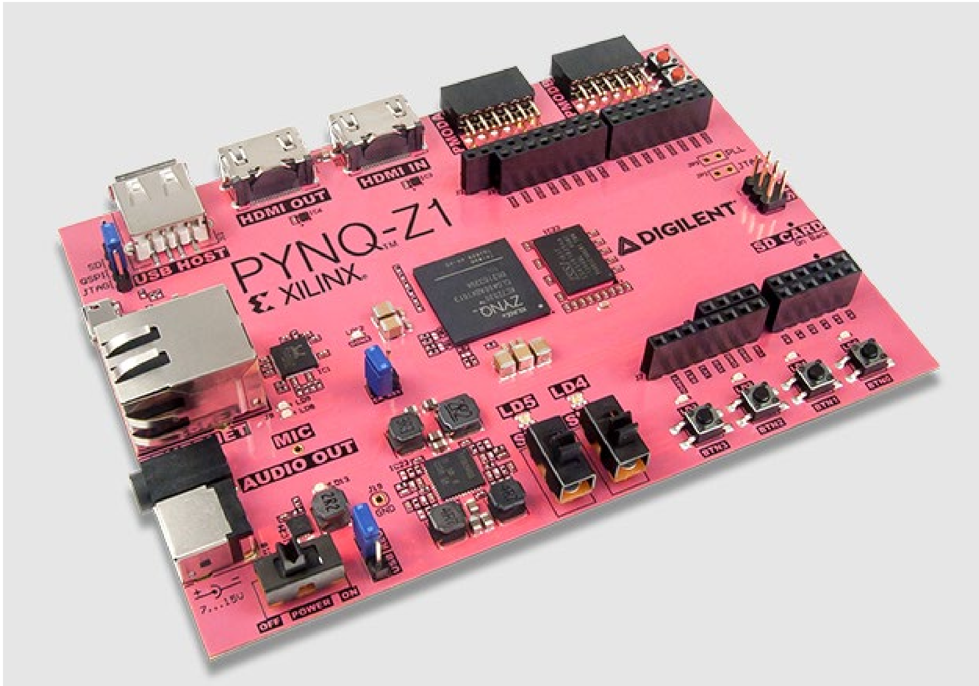


Figure 8. PYNQ - Z1.

Features:

- Fully supported by the PYNQ framework
- 650MHz dual-core Cortex-A9 processor
- 512 MB DDR3
- Wide range of USB, Ethernet, Video and Audio connectivity
- Arduino shield and Pmod connectors for adding-on hardware devices
- Programmable from JTAG, Quad-SPI flash, and microSD card

Key FPGA Specifications:

- Logic slices 13,300
- 6-input LUTs 53,200

- Flip-Flops 106,400
- Block RAM 630 KB
- DSP Slices 220
- Clock Resources Zynq PLL with 4 outputs
- 4 PLLs
- 4 MMCMs
- 125 MHz external clock
- Internal ADC Dual-channel, 1 MSPS

Electrical:

- Power Inputs USB
- 7V-15V External source

Section 4.2 - Vivado

Vivado[3] is a Xilinx design suite that provides a full set of tools from system-level design to hardware verification and implementation. Vivado tools include functions such as design entry, simulation, synthesis, placement and routing, and bitstream generation, which can help developers complete the FPGA design, verification, and generation processes. The Vivado tool provides a visual interface and a command line interface, supports various design methods and technologies, and enables designers to fully control and customize the FPGA.

[3] https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-sdk

Vitis High-Level Synthesis (Vitis HLS)[4] is a high-level synthesis tool launched by Xilinx, which allows developers to use high-level programming languages to describe hardware functions and convert them into hardware circuits on FPGA, simplifying FPGA Design flow that improves development speed and performance.

Section 4.3 - Docker

Docker[5] is an open source containerization platform for building, deploying and running applications. It provides a lightweight, portable and self-contained container environment that enables applications to run on different operating systems while providing good isolation and security.

In the project, we need to use FINN's docker to use FINN tool, and the required Brevitas and FINN libraries have been installed in FINN's docker. And use FINN's docker to export the ONNX file and convert it into a file that can be used on the PYNQ-Z1 board.

Section 4.4 - PuTTY

PuTTY[6] is a free, open-source SSH and Telnet client software for remote connection and communication between computers.

In this project, you need to use putty to connect to the PYNQ-Z1 development board.

[4] https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

[5] https://www.docker.com/

[6] https://www.putty.org/

Chapter5 Frameworks

In this project, we use Pytorch, Brevitas, Tensorflow to train the data set. Then use Vivado HLS, FINN tools and Tensil-AI tools to develop and deploy on the PYNQ-Z1 development board.

Section 5.1 - PyTorch

PyTorch[7] is a popular open-source deep learning framework that has gained significant attention in both research and practical applications. It provides a flexible and intuitive interface for designing, training, and deploying deep neural network models. With its dynamic computational graph and automatic differentiation capabilities, PyTorch enables researchers and developers to easily construct complex neural network architectures and perform efficient gradient-based optimization.

One of the key advantages of PyTorch is its ease of use and flexibility. The framework offers a wide range of pre-defined layers, loss functions, and optimization algorithms, making it straightforward to build and experiment with various neural network architectures. Additionally, PyTorch's dynamic nature allows for on-the-fly adjustments to the network structure and facilitates rapid prototyping and iterative model development.

Section 5.2 - BREVITAS

[7] https://pytorch.org/

Brevitas[8] is an open-source framework for quantized neural networks, designed to enable efficient neural network inference on hardware. It provides a set of tools and libraries that make the quantization and deployment of neural networks easier and feasible.

The core function of Brevitas is to quantize the neural network model, that is, to convert the weights and activation values in the model into low-bit representations. This quantization can significantly reduce the storage requirements and computational complexity of the model, enabling efficient inference on hardware. Brevitas supports various quantization methods, including fixed-point numbers and binarization, to meet different hardware platforms and application requirements.

In addition to quantization capabilities, Brevitas also provides a series of tools and optimization techniques for model pruning and quantization-aware training. Pruning is a technique that can improve inference efficiency by removing unimportant connections and neurons, reducing the number of parameters and calculations of the model. Quantization-aware training refers to considering the impact of quantization during training to improve the accuracy and performance of quantized models.

Brevitas is designed to integrate seamlessly with the PyTorch framework, so developers can use the familiar PyTorch API to build and train neural network models. Brevitas provides custom quantization modules that can be directly embedded into PyTorch models and work seamlessly with other PyTorch modules.

Section 5.3 - TENSORFLOW

[8] https://xilinx.github.io/brevitas/

TensorFlow[9] is an open source machine learning framework developed and maintained by the Google Brain team. It is widely used to build and train various types of machine learning models, including neural network models.

At the heart of TensorFlow is the concept of a computational graph, which is a way of representing computation as a graph structure. In TensorFlow, users can use Python or other supported programming languages to define a computational graph, which describes the flow of data and the operations between the data. This enables TensorFlow to efficiently perform complex computational tasks and utilize the structure of the computational graph for automatic derivation and optimization.

Section 5.4 - FINN

FINN (FPGA-INference) [5] is an open source framework designed to deploy neural network models to FPGA (Field-Programmable Gate Array) for efficient reasoning. It provides a set of tools and processes that make the process of quantifying, optimizing and deploying neural network models easier and more feasible. Through the FINN framework, users can convert the trained neural network model into a hardware description language (HDL) representation, and deploy and reason on the FPGA.

The core features of the FINN framework include Quantization and Optimization. Quantization technology converts the weights and activation values of neural network models into low-bit-width integer representations, thereby reducing the demand for computing and storage resources and improving the efficiency of hardware inference. Optimization technology improves the performance and efficiency of the neural network

[9] https://www.tensorflow.org/

model on the FPGA and reduces resource occupation through strategies such as parallel computing, layer fusion, and resource sharing.

The FINN framework also provides integration with the Xilinx Vivado toolset, making hardware design and bitstream generation easier. It supports a variety of Xilinx FPGA platforms, and provides some pre-trained neural network models as a starting point, which is convenient for users to quantify and optimize. By using the FINN framework, users can quickly and accurately deploy neural network models to FPGAs to achieve efficient hardware inference, which is suitable for applications in fields such as computer vision, embedded systems, and edge computing.

The following are the key features and components of the FINN framework:

1. Quantization:

   FINN supports the quantization of neural network models, converting floating-point weights and activation values into low-bit-width integer representations, thereby reducing storage and computing resource requirements. It provides a variety of quantization methods and techniques, including fixed-point quantization, binary quantization, and ternary quantization.

2. Optimization:

   The FINN framework improves the performance and efficiency of the neural network model on the FPGA through various optimization techniques, such as layer fusion, resource sharing, and pipeline. It also supports hardware optimizations such as memory optimization and memory pipelining to minimize FPGA resource usage and increase inference speed.

3. Deployment:

   FINN converts the quantized and optimized neural network model into a hardware description language (HDL) representation, which can be deployed on the FPGA. It provides integration with the Xilinx Vivado toolset, simplifying the hardware design and bitstream generation process.

4. Model Zoo (Model Library):

   The FINN framework provides some trained and optimized neural network models that can be quantified and optimized as a starting point. These models include classic convolutional neural networks (CNNs) models such as LeNet-5, AlexNet, and VGGNet, among others.

5. FPGA Targeting (FPGA target):

   FINN supports a variety of Xilinx FPGA platforms, including Zynq-7000, Zynq UltraScale+ MPSoC and Alveo series. Users can choose an FPGA platform that suits their application requirements and resource constraints for deployment.

By using the FINN framework, users can quickly and accurately deploy neural network models to FPGAs to achieve efficient hardware inference.

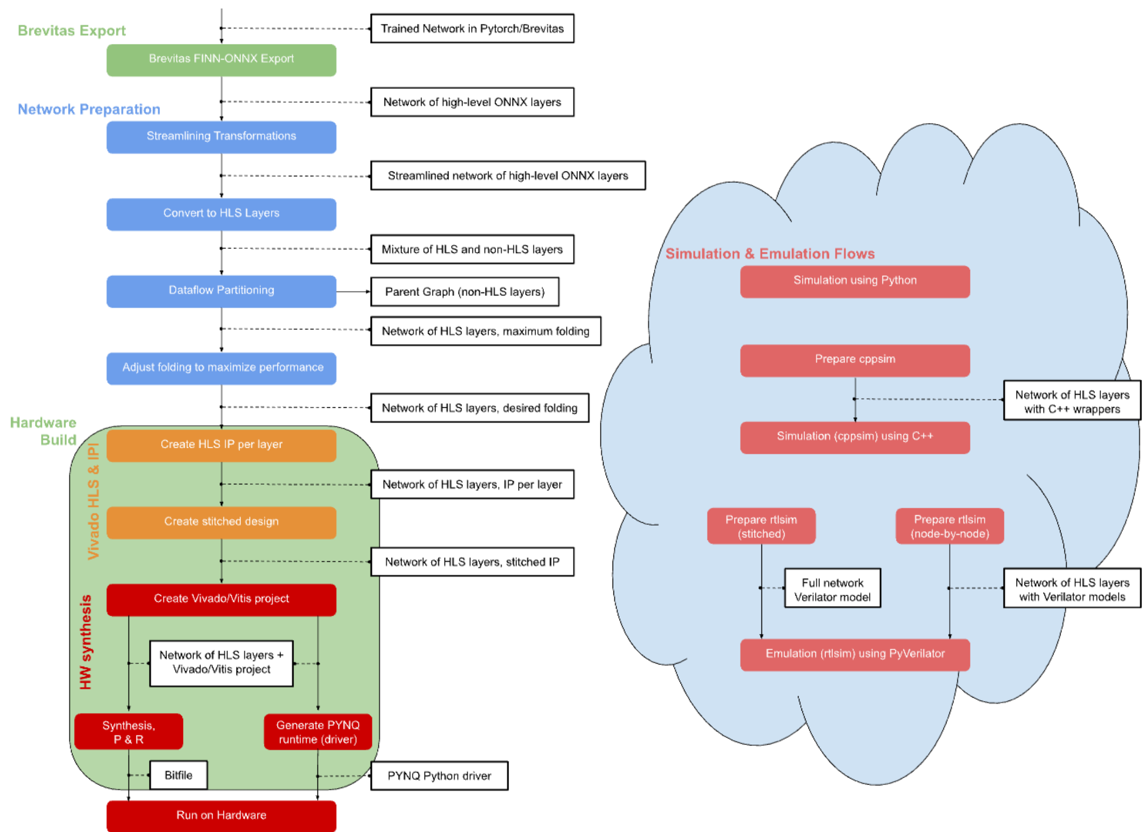


Figure 9. FINN design flow.

Section 5.4.1 - FINN-CNV

The FINN-CNV [6] model is a Convolutional Neural Network (Convolutional Neural Network) model based on the FINN (FPGA-INference) framework. FINN is an open source tool designed to deploy neural network models to FPGAs for efficient inference. The FINN-CNV model leverages the capabilities and optimization techniques of the FINN framework to enable the ability to deploy and infer convolutional neural networks on FPGAs.

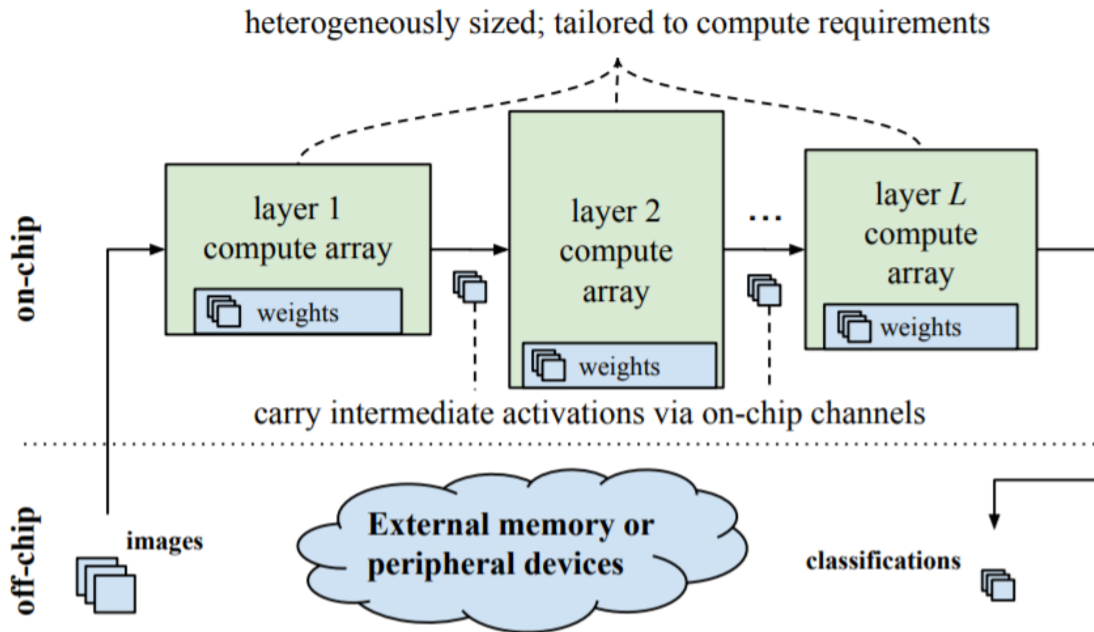


Figure 10. FINN hardware architecture.

On the one hand, the FINN-CNV model utilizes the parallel computing capability and low power consumption characteristics of FPGA, which can gain advantages in computer vision tasks with high real-time requirements. By converting the CNNs model into a hardware description language (HDL) representation, and performing quantization and optimization, the FINN-CNV model can achieve high-performance neural network reasoning with high computational efficiency and small model size.

On the other hand, the FINN framework provides a series of quantization and optimization techniques, making the FINN-CNV model flexible and scalable. Whether it is a simple network structure or a deep model, the FINN-CNV model can be deployed and optimized. Through custom hardware design and optimization strategies, users can customize the FINN-CNV model according to the needs of specific applications.

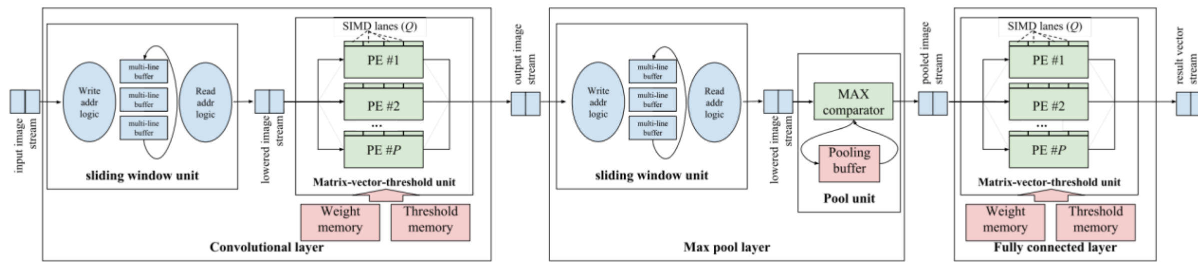


Figure 11. FINN application CNNs network.

The advantages of the FINN-CNV model include:

1. FPGA acceleration:

   By deploying the CNNs model to the FPGA, the parallel computing capability and low power consumption of the FPGA are used to achieve high-performance neural network reasoning. This makes the FINN-CNV model advantageous in scenarios with high real-time requirements, such as real-time object detection and video processing.

2. Quantization and optimization:

   The FINN framework provides a series of quantization and optimization techniques that can reduce the accuracy of the CNNs model and reduce the demand for computing and storage resources by quantizing weights and activation values. This enables the FINN-CNV model to have higher computational efficiency and smaller model size while maintaining high inference performance.

3. Scalability and flexibility:

The FINN-CNV model can adapt to CNNs models of different scales and complexity, and can be deployed and optimized from simple network structures to deep models. The FINN framework provides flexible interfaces and tools for custom hardware design and optimization strategies to meet the needs of specific applications.

Overall, the FINN-CNV model provides a high-performance, low-power solution through quantization and optimization techniques while utilizing FPGA acceleration. It is suitable for application scenarios with high real-time requirements in computer vision tasks, and has flexibility and scalability, and can adapt to CNNs models of different sizes and complexities. With the support of the FINN framework, users can deploy and optimize convolutional neural network models more conveniently to achieve efficient neural network reasoning.

Section 5.5 - Tensil-AI

Tensil [10] is a machine learning model compiler and hardware generator that enables you to create and deploy an optimized custom ML inference accelerator for your application. We can choose the architecture of PYNQ, and then deploy the neural network. Use the Tensil tool to generate an overplay file that PYNQ can use, and then perform inference on PYNQ.

Chapter6 Methodology and Experiments

Section 6.1 - Methodology

In this project, I will use the following method:

[10] https://www.tensil.ai/

FPGA Development Board Preparation: The first phase involves acquiring the designated FPGA development board for the project. Detailed information regarding the board's functionality, capabilities, and specifications will be thoroughly reviewed. Furthermore, the necessary development board environment will be installed to ensure seamless integration with the project requirements.

Algorithmic Content Preparation: This step encompasses gathering the necessary data and content required for the algorithmic deployment. Careful consideration will be given to selecting appropriate datasets and acquiring the relevant algorithmic environment. This ensures compatibility and optimization of the deployment process.

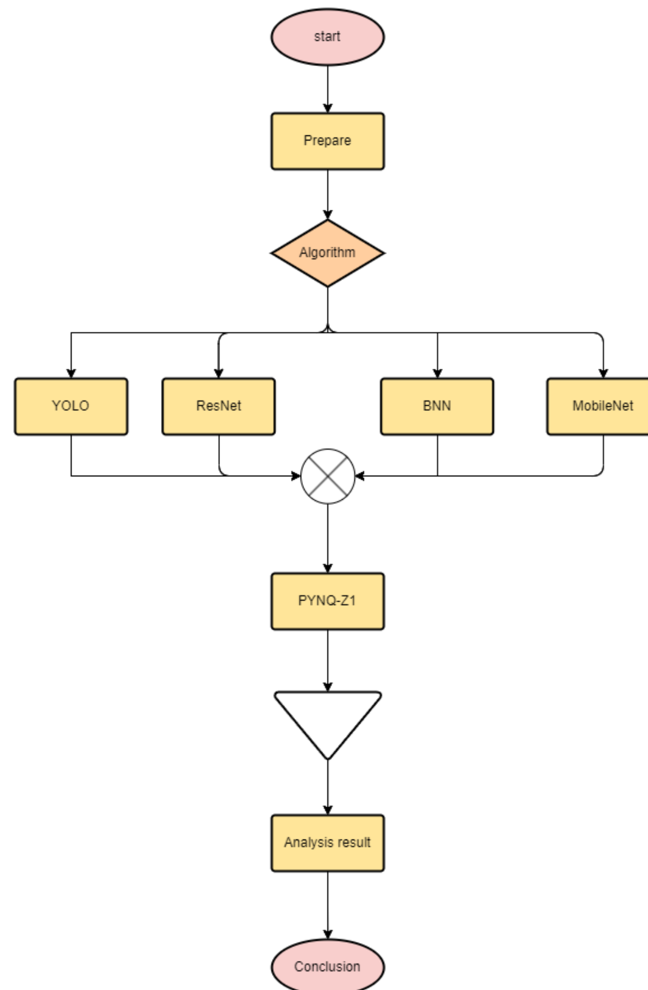
Algorithm Deployment: The deployment phase will involve implementing the selected algorithm onto the prepared development board. This will entail configuring the necessary settings, initiating the deployment process, and ensuring the successful execution of the algorithm on the FPGA platform.

Result Evaluation and Comparison: The obtained results will be thoroughly examined and assessed. A comprehensive analysis will be conducted, comparing the deployed algorithm's performance and outcomes against predefined benchmarks or baseline metrics. The evaluation process will encompass aspects such as accuracy, speed, efficiency, and any other relevant performance indicators.

Conclusion and Findings: Based on the evaluation and comparison results, a conclusive summary will be formulated. This conclusion will provide insights into the effectiveness and feasibility of the deployed algorithm on the FPGA development board.

Figure 12. Flow Chart.

Furthermore, any significant findings, limitations, or areas for future exploration will be highlighted to contribute to the existing knowledge in the field.



Section 6.2 - Synthesis results

Since our FPGA board is PYNQ-Z1, its resources are very limited, so the neural networks we deploy are all quantized. Among them, Yolov2, ResNet20, and MobileNetv2 all use 16-bit quantization. Due to the particularity of the structure of BNN, we use one-bit weight and one-bit activation.

Section 6.2.1 - COCO dataset

The whole idea is to customize the generation of IP cores on Vitis HLS, and then generate BIT stream files on Vivado, and then transfer them to PYNQ-Z1 for operation.

We use YOLO in Tensorflow for neural network deployment and using the COCO dataset.

To use Vitis HLS and Vivado, you need to find the board configuration file[11] of PYNQ-Z1. Here we can find the board configuration file of the Xilinx development board on PYNQ's official website.

Section 6.2.1.1 - YOLOv2 model

YOLOv2 contains convolutional layers, pooling layers, routing layers and reordering layers. The convolutional layer extracts features, the pooling layer is used for pixel sampling, the routing layer is used for multi-level feature fusion, and the reordering layer is used for sampling rearrangement of features.

1) FPGA-Based Optimization

While maintaining the same accuracy, the resource consumption of calculation can be reduced by reducing the data bit width. And as the data bit width decreases, the amount of transmitted data also decreases. We use 16-bit quantization.

---

[11] https://pynq.readthedocs.io/en/v2.7.0/overlay_design_methodology/board_settings.html#vivado-board-files

The convolutional layer occupies most of the calculation of the model, and FPGA can be used to accelerate the convolution operation. The pooling layer is similar to the convolutional layer, so the processing of the pooling layer and the convolutional layer is placed on the FPGA side. [9]

2) HLS synthesis

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1242	-
FIFO	-	-	-	-	-
Instance	139	110	32817	40360	0
Memory	65	-	0	0	0
Multiplexer	-	-	-	3671	-
Register	-	-	2964	-	-
Total	204	110	35781	45273	0
Available	280	220	106400	53200	0
Utilization (%)	72	50	33	85	0

Figure 13. Report HLS synthesis.

72% usage of BRAM_18K: BRAM_18K is a memory cell in the FPGA that is used to store and read data. This means that in the design, most of the data storage requirements are occupied by BRAM_18K. High BRAM_18K usage may indicate that a design needs to process large amounts of data or use large amounts of storage.

The usage rate of DSP48E is 50%: DSP48E is a digital signal processor in FPGA, which provides operation functions such as multiplication, addition, subtraction, and accumulator. It has highly parallel capabilities, high-speed data paths, and low-latency computing capabilities. A utilization of 50% indicates that the DSP48E resources are fully utilized in the design and heavy digital signal processing operations are required.

FF usage is 33%: FF (Flip-Flop) is used to store state information. A usage rate of 33% means that there are relatively few state variables or state machines in the design.

The usage rate of LUT is 85%: LUT (Lookup Table) is used to implement logical operations, such as AND, OR, NOT, XOR, etc. A utilization rate of 85% indicates that the LUT resource is fully utilized in the design, which may contain a large number of logic operations and complex digital logic circuits.

3) ***Block Design***

Import the IP core to the Vivado project.

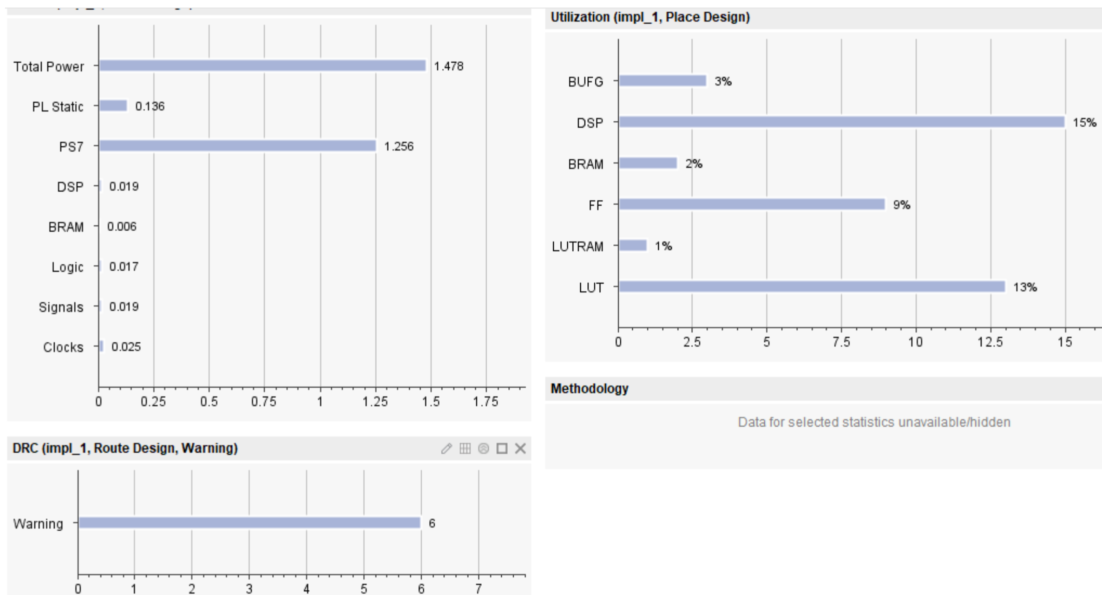


Figure 14. Vivado – Project -information-2.

Note the reduced power consumption, less than 1.5W, shown in Figure 14. Also, we can view the block design as shown in Figure 15. To have an idea of what this means, just let's take into account that typically the power consumption of general purpose processors is in the order of tens of watts.

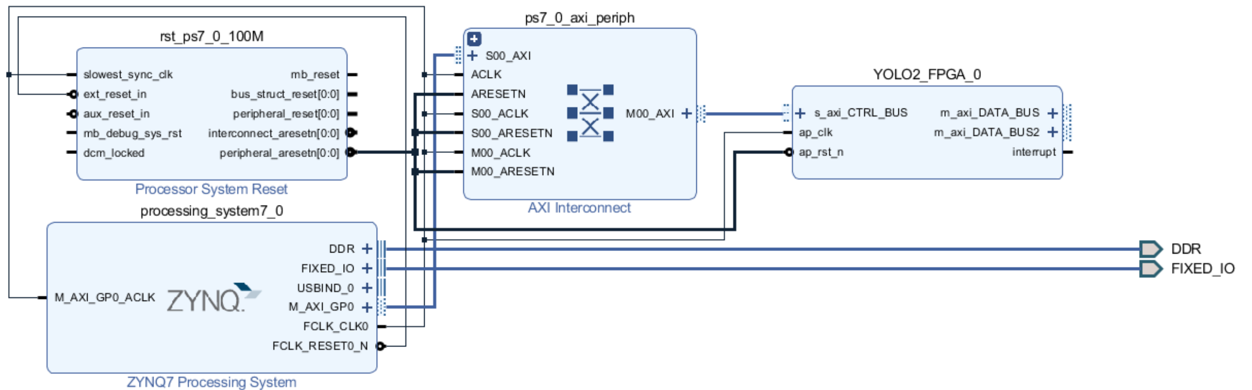


Figure 15. Block-design Diagram.

Finally, after we generate the BIT stream file, we can transfer the BIT stream file to the PYNQ-Z1 board and use jupyter-book and python to call yolo for image recognition.

We can open the device design to view.

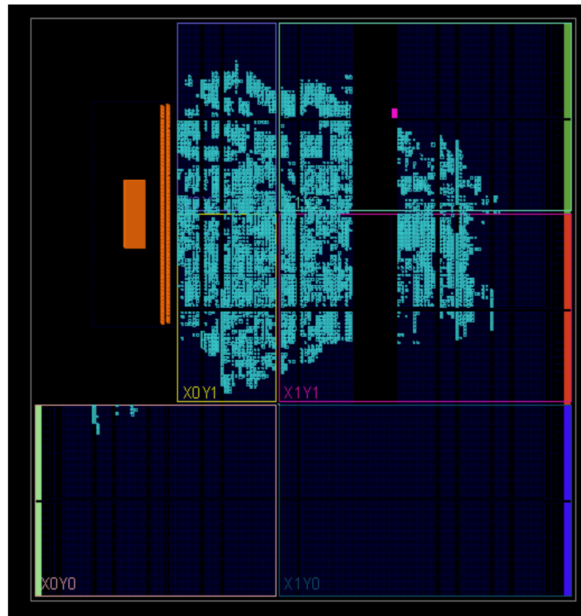


Figure 16. Implemented Design with Yolov2.

Section 6.2.2 - CIFAR-10 dataset

Section 6.2.2.1 - XNORNet

Here we evaluate XNORNet as an example of BNN to perform classification, and we use Brevitas training files in ONNX format. Then use a FINN tool to convert it to an HLS layer. After creating the HLS IP, we convert it to a file that can be used and run on PYNQ. All weights and activations in the network models [6] we use are quantized to bipolar values (-1 and +1), except for inputs and outputs.

```
import onnx
from finn.util.test import get_test_model_trained
import brevitas.onnx as bo
from qonnx.core.modelwrapper import ModelWrapper
from qonnx.transformation.infer_shapes import InferShapes
from qonnx.transformation.fold_constants import FoldConstants
from qonnx.transformation.general import GiveReadableTensorNames, GiveUniqueNodeName

cnv = get_test_model_trained("CNV", 1, 1)
bo.export_finn_onnx(cnv, (1, 3, 32, 32), build_dir + "/end2end_cnv_w1a1_export.onnx")
model = ModelWrapper(build_dir + "/end2end_cnv_w1a1_export.onnx")
model = model.transform(InferShapes())
model = model.transform(FoldConstants())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(RemoveStaticGraphInputs())
model.save(build_dir + "/end2end_cnv_w1a1_tidy.onnx")
```

Figure17. Trained-Model into ONNX.

This is to convert the trained model into an HLS layer using the FINN tool.

We can see structure of ONNX:

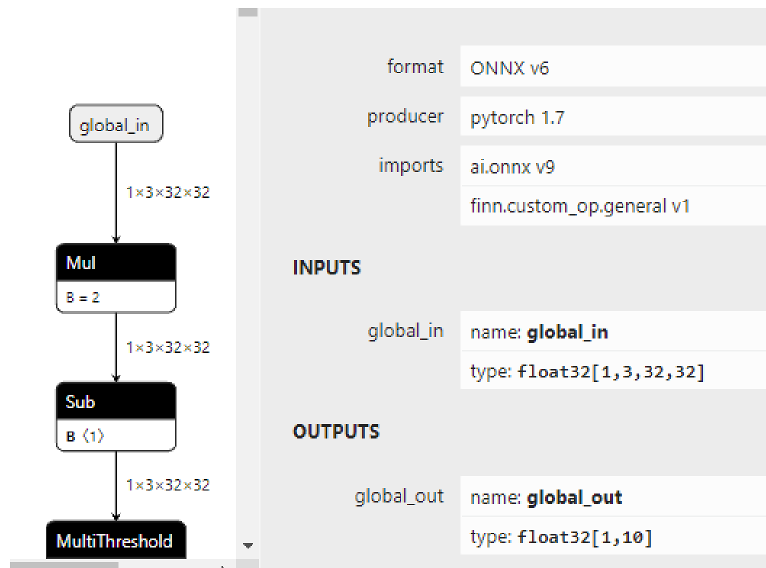


Figure 18. Model ONNX original.

This is the input of the ONNX model when the FINN tool is not used.

We need to make a little adjustment to the network structure, limit the input of the network between [0,1], and the output is the highest probability. Then review the structure of onnx.

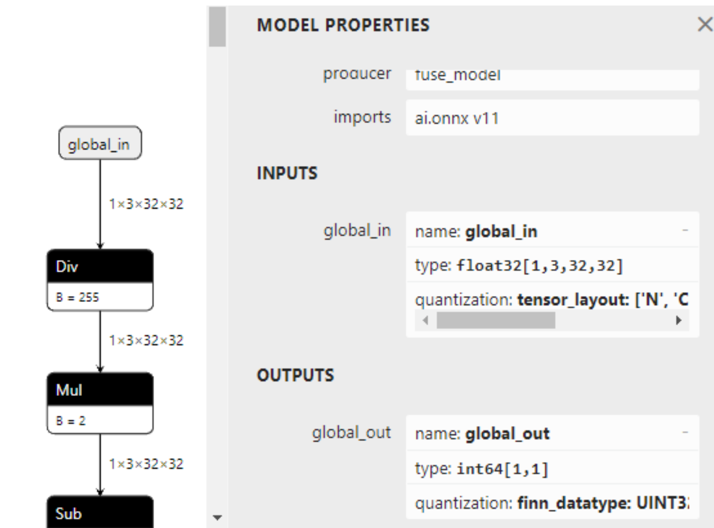


Figure 19. Model ONNX pre and post processing.

This is followed by implementing the convolution in FINN (Figure[17]), here we turn it into a matrix multiplication operation. We use streamlined conversion here, which can make a few bit activations become thresholds, so that floating point can not be used.

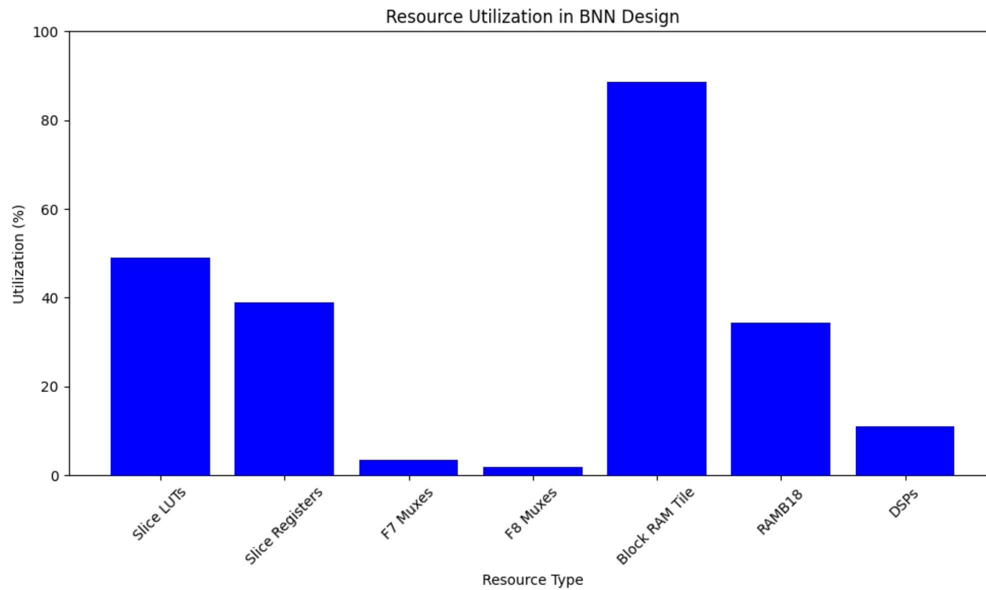


Figure 20. Synthesis information.

The specified time constraints are met. The resource utilization of Block RAM Tile is very high at 88.57%. Block RAM is used extensively in the design, usually for storing large data sets or applications that require high-performance storage.

Then it can be converted to HLS layer.

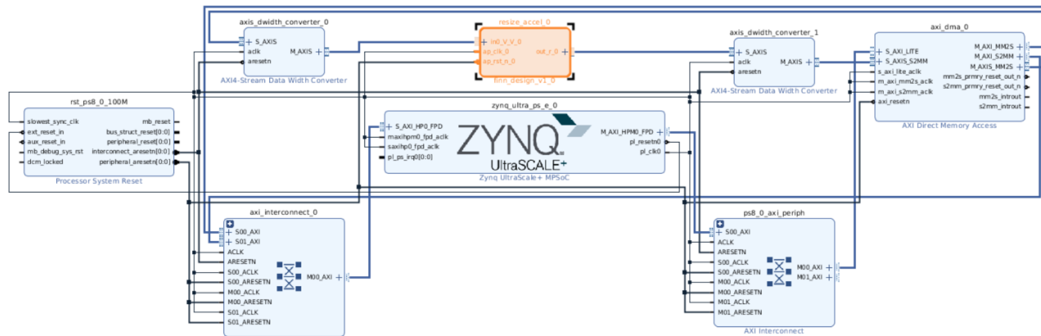


Figure 21. Block-design Diagram.

Figure 21 is a composite diagram in which the module is synthesized on the PYNQ-Z1 development board.

We also transfer the bit file to PYNQ-Z1 for deployment.

Section 6.2.2.2 - ResNet

I use the pre-trained ONNX model to deploy ResNet, and use the Tensil tool to convert it into a file that can be deployed on PYNQ. Before converting the weight file, it is also necessary to deploy the Vivado.

Table 1 is the information automatically exported after using the Tensil-AI tool.

Category	Value
Model	resnet20v2_cifar_onnx_pynqz1
Data type	FP16BP8
Array size	8
DRAM0 memory size (vectors/scalars/bits)	1,048,576 / 8,388,608 / 20
DRAM1 memory size (vectors/scalars/bits)	1,048,576 / 8,388,608 / 20
Local memory size (vectors/scalars/bits)	8,192 / 65,536 / 13
Accumulator memory size (vectors/scalars/bits)	2,048 / 16,384 / 11
Stride #0 size (bits)	3
Stride #1 size (bits)	3
Operand #0 size (bits)	16
Operand #1 size (bits)	24
Operand #2 size (bits)	16
Instruction size (bytes)	8
DRAM0 maximum usage (vectors/scalars)	26,624 / 212,992
DRAM0 aggregate usage (vectors/scalars)	103,458 / 827,664
DRAM1 maximum usage (vectors/scalars)	71,341 / 570,728
DRAM1 aggregate usage (vectors/scalars)	71,341 / 570,728
Local memory maximum usage (vectors/scalars)	8,192 / 65,536
Local memory aggregate usage (vectors/scalars)	393,294 / 3,146,352
Accumulator memory maximum usage (vectors/scalars)	2,048 / 16,384
Accumulator memory aggregate usage (vectors/scalars)	199,212 / 1,593,696
Maximum number of stages	16
Maximum number of partitions	24
Execution latency (MCycles)	1.916
Aggregate latency (MCycles)	1.916
Execution energy (MUnits)	110.019
Aggregate energy (MUnits)	110.020
MAC efficiency (%)	50.144
Total number of instructions	237,189
Compilation time (seconds)	28.332
True consts scalar size	568,466
Consts utilization (%)	97.545
True MACs (MMAC)	61.476
MAC efficiency (%)	50.144

Table 1. ResNet20 compiler summary.

In the compiler summary, the efficiency of MAC (Multiply-Accumulate) is 50.144%. This means that when performing calculations, only about half of the MAC operations are actually used to perform calculation tasks, and the other half may not be effectively utilized or not executed.

Section 6.2.2.3 - MobileNetv2

We use MobileNetv2 to train the CIFAR-10 dataset, then export the weight file to ONNX format, and use the Tensil tool to compile the model. Note that since the input size of MobileNetv2 itself is 224×224, and the image size of the CIFAR-10 dataset is 32×32, I adjusted the model appropriately. The remaining steps are the same as above.

Table 2 shows the information automatically exported after using the Tensil-AI tool.

Category	Value
Model	mobilenet_onnx_pynqz1
Data type	FP16BP8
Array size	8
DRAM0 memory size (v/s/b)	1,048,576 / 8,388,608 / 20
DRAM1 memory size (v/s/b)	1,048,576 / 8,388,608 / 20
Local memory size (v/s/b)	8,192 / 65,536 / 13
Accumulator memory size (v/s/b)	2,048 / 16,384 / 11
Stride #0 size (bits)	3
Stride #1 size (bits)	3
Operand #0 size (bits)	16
Operand #1 size (bits)	24
Operand #2 size (bits)	16
Instruction size (bytes)	8
DRAM0 maximum usage (v/s)	1,220 / 9,760
DRAM0 aggregate usage (v/s)	1,298 / 10,384
DRAM1 maximum usage (v/s)	8,127 / 65,016
DRAM1 aggregate usage (v/s)	8,127 / 65,016
Local memory maximum usage (v/s)	6,080 / 48,640
Local memory aggregate usage (v/s)	9,697 / 77,576
Accumulator memory maximum usage (v/s)	1,764 / 14,112
Accumulator memory aggregate usage (v/s)	2,298 / 18,384
Maximum number of stages	1
Maximum number of partitions	1
Execution latency (KCycles)	54.748
Aggregate latency (KCycles)	54.768
Execution energy (MUnits)	2.714
Aggregate energy (MUnits)	2.714
MAC efficiency (%)	18.600
Total number of instructions	13,532
Compilation time (seconds)	3.647
True consts scalar size	62,006
Consts utilization (%)	79.119
True MACs (KMAC)	651.720
MAC efficiency (%)	18.600

Table 2. MobileNetv2 compiler summary.

In the compiler summary, the efficiency of MAC (Multiply-Accumulate) is 18.6%. This means that when performing calculations, the performance of the hardware system is very low and a lot of hardware resources are wasted.

Figure 22 shows the Synthesize for PYNQ-Z1 before using the Tensil-AI tool. Guaranteed to run on PYNQ-Z1.

Drag tensil's module onto the module design diagram, and then click "Run Block Automation" and "Run Connection Automation". Add "AXI SmartConnect" again. We need 4 more. First 3 instances (smartconnect_0 to smartconnect_2) are necessary to convert AXI version 4 interfaces of the TCU and the instruction DMA block to AXI version 3 on the PS. The smartconnect_3 is necessary to expose DMA control registers to the Zynq CPU, which will enable software to control the DMA transactions.

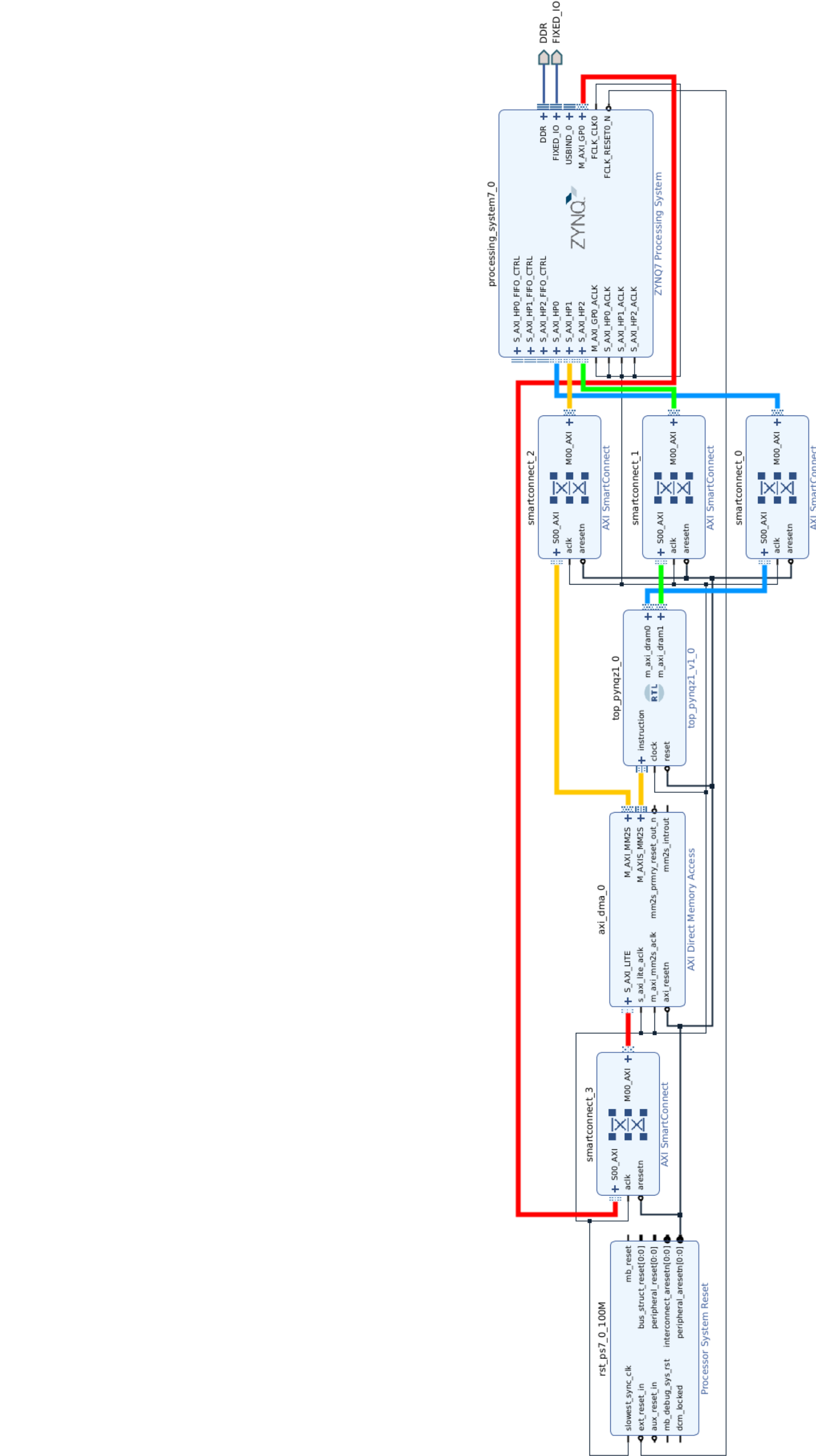


Figure 22. Block-design Diagram-Tensil.

Section 6.3 - Execution time

Section 6.3.1 - Time for each network to run on PYNQ using the FPGA accelerator:

CPU: i7-9700

FPGA: PYNQ-Z1

	Yolov2	XNORNet	Resnet20	MobileNetv2
CPU	0.69s	1.59s	0.016s	0.017s
FPGA	1.05s	0.0015s	0.127s	0.02145s

Table 3. Times in PYNQ with FPGA's accelerator.

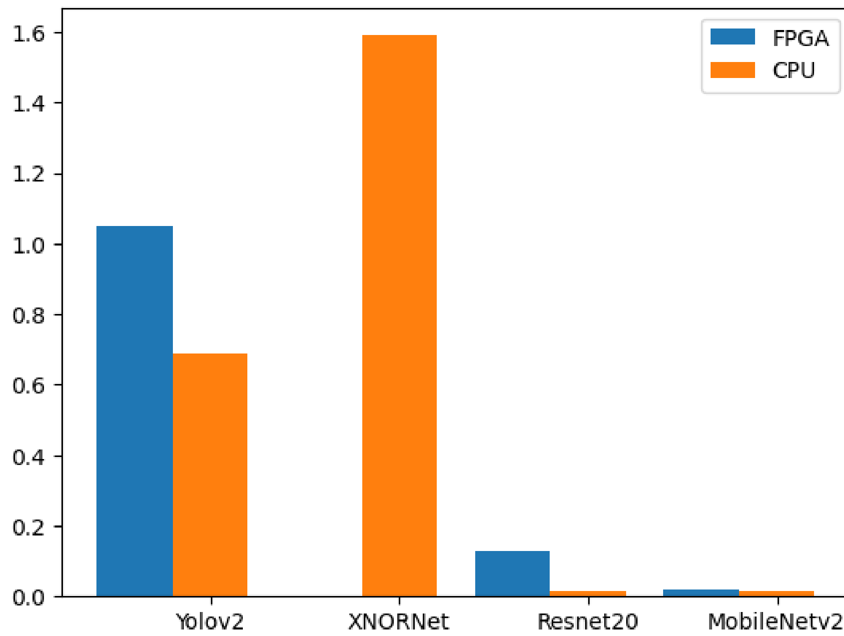


Figure 23. Times in PYNQ with FPGA's accelerator.

It should be noted that since there is no corresponding environment on PYNQ-Z1, it is the inference time running on the CPU with the environment.

1. For the same 16-bit quantization, the inference time of Yolov2 is about 8 times slower than that of ResNet20, and about 50 times slower than that of MobileNetv2. The same meaning is that ResNet20 is about 6 times slower than MobileNetv2 inference speed. This result is actually expected, because from the size of the network model, Yolov2 is larger than ResNet20 and MobileNetv2.

2. For the same data set CIFAR-10, the reasoning speed of BNNs is the fastest among them. There may be several reasons:

2.1. BNNs have the highest degree of quantization, using 1-bit weight, which is a very extreme quantization method.

2.2. BNNs are network models specific to PYNQ-z1, and converted into accelerators that can be deployed on PYNQ-z1 using the FINN tool.

3. The execution time of Yolov2 on the FPGA is longer. This is because only the convolution and pooling are being executed on the FPGA, while the other operations are performed on the ARM core within the Z1. Thus, there is data transfer between the ARM core and the FPGA. This causes the execution time of Yolov2 on the FPGA to be slower.

Section 6.4 - Accuracy

Section 6.4.1 - Top-1 Accuracy for each network to run on PYNQ using the FPGA accelerator:

Yolov2 is generally a deep detection model for target detection, but most of us here are focused on image classification tasks. Therefore, we will not discuss mAP here; instead, we will only focus on top-1 accuracy, with the inner Darknet-19 model performing classification.

	Yolov2	XNORNet	Resnet20	MobileNetv2
FP32	76.50%	94.27%	95.37%	94.03%
FP16	73.30%	83.13%	92.06%	76.13%

Table 4. Accuracy in PYNQ with FPGA's accelerator.

The colored red line represents the accuracy using FP32.

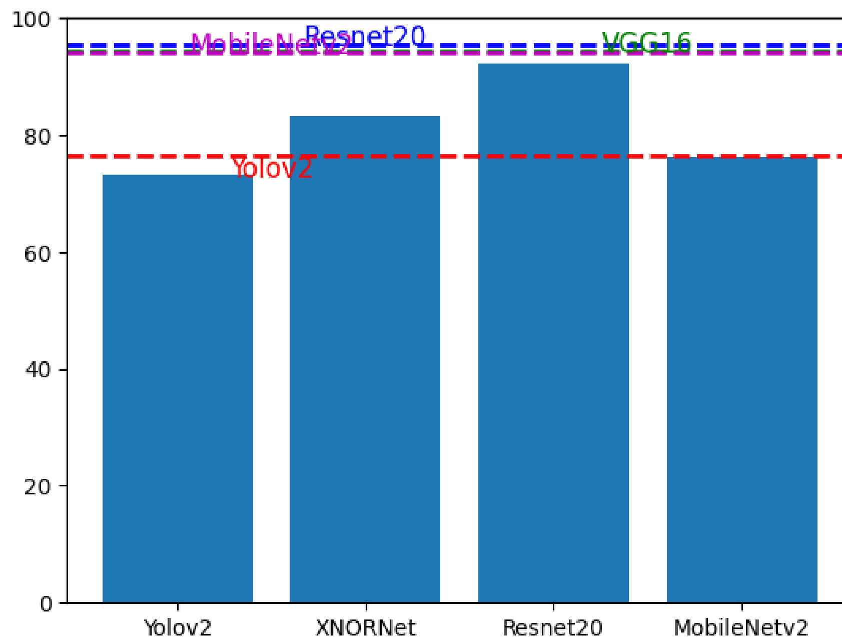


Figure 24. Accuracy in PYNQ with FPGA's accelerator.

For the same data set CIFAR-10 data set, ResNet20 has the highest accuracy rate, reaching 92.06%. ResNet20 itself is a model with high performance in image classification tasks. Although the accuracy rate of XNORNet is only 83.13%, I think it has an advantage in the resource-constrained PYNQ-Z1, because its speed is 84 times that of ResNet20. The accuracy of MobileNetv2 is the lowest. Although it is designed for resource-constrained environments, it may not perform as well as other models for the CIFAR-10 dataset.

Chapter8 Conclusions and future work

Section 8.1 - Conclusions

Based on our task completion and results, we draw some conclusions:

1. In this work I deployed a YOLO network, a BNN network, a ResNet network and a MobileNetv2 network on PYNQ-Z1, and compared their inference time and accuracy for their data sets.

2. The results seen are that it is possible to deploy traditional DNN networks, but except for FPGA-specific customized networks, other networks have good performance, but are not particularly outstanding. And due to the resource limitations of PYNQ-Z1, a quantified version needs to be deployed, which has a certain impact on the performance loss of the model.

3. For PYNQ-Z1, because resources are very limited, many networks cannot be directly deployed on it. I have tried many models, but basically "Insufficient DRAM memory to allocate" will appear.

4. For model transformation, RTL language is traditionally used for programming, which is actually a very difficult task, but using Python and existing tools, the time to deploy DNN models is reduced.

5. For existing tools, such as FINN and Tensil used this time, there are still restrictions on the use of FINN and Tensil. FINN must use Brevitas quantization training to convert, and there will be some unpredictable problems during the conversion. Because some layers cannot be perfectly converted during HLS conversion. However, using the Tensil tool must accurately define the hardware of the FPGA board.

The results of the experiment actually reflect a very common problem, which is to weight the relationship between network size, precision and accuracy.

To sum up, we have completed the task. We deploy quantized versions of four neural networks on FPGAs, and discuss network size, accuracy, and inference time. For neural network models that are highly customized to FPGAs, the advantages are still very large, especially for the actual situation that requires inference speed.

Section 8.2 - Future Work

In the future, there are still great challenges in deploying neural network models on FPGAs. The following are possible research directions and improvements:

1. Improved performance: FPGA has the potential to accelerate neural network reasoning, but is limited by hardware resources, and new hardware architectures and design methods can be researched.

2. Algorithm optimization: In addition to hardware improvements, research into more efficient neural networks. Use quantization and pruning techniques to reduce computing and storage resource requirements, and improve the performance and efficiency of neural networks on FPGAs.

3. Automated tools and processes: Simplifying the deployment process is critical for the widespread adoption of neural network models on FPGAs. Automated tools and processes can be developed to simplify the steps of model conversion, compilation and deployment, improve developer efficiency and lower the barriers to deployment.

4. Flexibility and scalability: In order to adapt to different application requirements, the flexibility and scalability of deploying neural network models on FPGA can be studied. This includes supporting different network architectures, model sizes, and accuracy requirements, and providing flexible interface and configuration options to meet the needs of different application scenarios.

By solving these problems, we can better utilize the advantages of FPGAs to achieve efficient and low-power neural network inference, and promote the development of hardware acceleration in the field of artificial intelligence.

REFERENCES

[1] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.

[2] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Communications of the ACM, 2017, 60(6): 84-90.

[3] Redmon J, Farhadi A. YOLO9000: better, faster, stronger[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2017: 7263-7271.

[4] Courbariaux M, Hubara I, Soudry D, et al. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1[J]. arXiv preprint arXiv:1602.02830, 2016.

[5] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott and M. Platzner, "RadioML Meets FINN: Enabling Future RF Applications With FPGA Streaming Architectures," in IEEE Micro, vol. 42, no. 6, pp. 125-133, 1 Nov.-Dec. 2022, doi: 10.1109/MM.2022.3202091.

[6] Umuroglu Y, Fraser N J, Gambardella G, et al. Finn: A framework for fast, scalable binarized neural network inference[C]//Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays. 2017: 65-74.

[7] Krizhevsky A, Hinton G. Learning multiple layers of features from tiny images[J]. 2009.

[8] Lin T Y, Maire M, Belongie S, et al. Microsoft coco: Common objects in context[C]//Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13. Springer International Publishing, 2014: 740-755.

[9] Ma Y, Cao Y, Vrudhula S, et al. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks[C]//2017 27th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2017: 1-8.

[10] Mas J, Panadero T, Botella G, et al. CNN Inference acceleration using low-power devices for human monitoring and security scenarios[J]. Computers & Electrical Engineering, 2020, 88: 106859.

[11] V. Sze, Y. -H. Chen, T. -J. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017, doi: 10.1109/JPROC.2017.2761740.

[12] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida and N. Bagherzadeh, "Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks," in IEEE Transactions on Computers, vol. 68, no. 5, pp. 660-675, 1 May 2019, doi: 10.1109/TC.2018.2880742.

[13] M. S. Kim, A. A. Del Barrio, H. Kim and N. Bagherzadeh, "The Effects of Approximate Multiplication on Convolutional Neural Networks," in IEEE Transactions on Emerging Topics in Computing, vol. 10, no. 2, pp. 904-916, 1 April-June 2022, doi: 10.1109/TETC.2021.3050989.

[14] Kim H J, Shin J, Del Barrio A A. Ctmq: Cyclic training of convolutional neural networks with multiple quantization steps[J]. arXiv preprint arXiv:2206.12794, 2022.

[15] Wang C Y, Bochkovskiy A, Liao H Y M. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023: 7464-7475.

[16] Everingham M, Van Gool L, Williams C K I, et al. The pascal visual object classes (voc) challenge[J]. International journal of computer vision, 2010, 88: 303-338.

[17] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.

[18] Sandler M, Howard A, Zhu M, et al. Mobilenetv2: Inverted residuals and linear bottlenecks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 4510-4520.

[19] Rastegari M, Ordonez V, Redmon J, et al. Xnor-net: Imagenet classification using binary convolutional neural networks[C]//European conference on computer vision. Cham: Springer International Publishing, 2016: 525-542.

[20] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.