A Dynamic Configurable FPGA Implementation of YOLOv3-tiny

Zhewen Yu
B.Eng in Electronic Information Engineering
Nanjing Tech Univeristy, 2018

Supervised by:
Dr Christos-Savvas Bouganis

A Thesis submitted in fulfilment of requirements for the degree of
Master of Science
Analogue and Digital Integrated Circuit Design
of Imperial College London

Department of Electrical and Electronic Engineering
Imperial College London
September 4, 2019

Abstract

Convolution Neural Network (CNN) has gained its favour in object detection. In order to deploy CNNs in embedded applications, low latency and low power consumption are desired. FPGAs are suitable for handling such tasks. There are many CNN frameworks proposed. YOLOv3-tiny is a light-weight network, achieving a great balance between precision and network complexity.

This thesis focuses on a dynamic configurable FPGA architecture for YOLOv3-tiny. The dynamic configurability means a unified accelerator is proposed. By setting typology parameters of the hardware accelerator during run-rime, different layer descriptions can be dealt with.

Individual IPs for different layer types are firstly designed. Fixed point quantisation and channels interleaving are main optimisations adopted. IPs are tunable through design parameters which control parallelism. Based on design parameters and network typology parameters, models are built for resources and latency estimations. In terms of latency, an extra software latency model is included to capture costs on the processor.

Based on models, Design Space Exploration (DSE) is carried out showing trade-off between resources and latency. Within the design space, optimal parameter choices reach lowest latency with as few resources as possible. The exploration is also constrained by resources available and test platforms. Finally, a design example on Zedboard is evaluated. It achieves 1.88 FPS on a 16-bit fixed point implementation, and 30.9% mAP50 on COCOval5k dataset.

Acknowledgment

First, I want to show my special gratitude to my supervisor Dr Christos-Savvas Bouganis. You offered me great freedom on choosing this topic for the project. Your helpful advice and guidance pushed me further on design methodology and optimisation, which formulated current work. I am grateful for our thorough analysis on project details, and it helped me understand many interesting findings better.

I also like to thank Hoi Chon Chao and Bo Wen for our discussions on implementing convolutional neural networks. Although we focused on different targets and used different platforms, their kind sharing kept me open-minded and exposed me to lots of possibilities.

Finally, I would like to thank my parents for their continuous support during the year. Their encouragement and company through video phones made me optimistic and determined in front of stress and challenges.

Contents

List of Figures

List of Tables

Abbreviations

DNN:	Deep Neural Network
GPU:	Graphics Processing Unit
FPGA:	Field Programmable Gate Array
HLS:	High-level Synthesis
FLOPS:	Floating-point Operations Per Second
ReLU:	Rectified Linear Unit
RoI:	Region of Interest
FPS:	Frames Per Second
mAP:	mean Average Precision
FB:	Full Buffering
PB:	Partial Buffering
DSE:	Design Space Exploration
MAC:	Multiply Accumulate
DSP:	Digital Signal Processing
GeMM:	General Matrix-Matrix Multiplication
SIMD:	Single Instruction Multiple Data
SVM:	Support Vector Machine
DDR:	Double Data Rate SDRAM
DMA:	Direct Memory Access
AXI:	Advanced eXtensible Interface
BN:	Batch Normalisation
FIFO:	First-In First-Out

RTL:	Register Transfer Level
BRAM:	Block Random Access Memory
FF:	Flip Flop
LUT:	LookUp Table
II:	Initial Interval
PS:	Processing System
PL:	Programmable Logic
NMS:	Non Maximum Suppression
GOPS:	Giga Operations Per Second

Chapter 1

Introduction

1.1 Motivation

IMAGE processing is a key technology in the information age. It provides electronic systems with the ability to sense, analyse and shape the world. Traditional image processing techniques mainly focus on certain mathematics algorithm or feature descriptor [4]. Since the beginning of this century, bio-inspired deep neural networks (DNNs) have gained their favour.

The main challenge of DNNs comes from the long computation time. The intensive arithmetical operations require platforms with high processing capability. Under most circumstances, computation ability corresponds to power consumption. Typical examples are GPUs or large-scale cloud servers.

In order to deploy neural networks in embedded applications, there are many researches carrying out to balance performance with power consumption. In terms of platforms, FPGAs appear to be promising, as the ability of hardware reconfiguration enables flexible parallelism [5]. Many hardware acceleration tricks make FPGA suitable for energy-efficient scenarios.

Recently, various network architectures have been proposed for object detection which can be very useful to develop intelligent systems. Putting these networks on FPGA is relatively challenging because of larger network scale. In addition, it is desirable that a tunable FPGA implementation can be provided for these complex networks. Design decisions can be made for different application requirements without building the system from scratch repeatedly.

1.2 Contribution

This work gives a FPGA implementation of YOLOv3-tiny neural network via Xilinx HLS tools. FPGA hardware IPs constitute a CNN accelerator. Based on the network typology and data analysis, fixed-point quantisation and channels interleaving are adopted to boost acceleration.

A dynamic configurable architecture is finally given, which computes all layers in the YOLOv3-tiny typology without bitstream reconfiguration. The method of network transformation is applied. It reshapes original network structure in order to be consistent with the accelerator.

Performance model and resources estimation of the whole system are provided. Design Space Exploration (DSE) is then performed to evaluate all valid design point. By choosing different design parameters, the architecture is tunable and can be deployed on almost any Xilinx ZYNQ device.

An optimal design for Zedboard platform with ZYNQ-7020 is demonstrated. The design is compared with similar works in terms of target networks, platforms, speed, resource and power. Finally, the FPGA implementation shows its advantage over CPU and GPU.

Just to clarify, design parameters mean those affect hardware details, like number of memory banks, maximal number of channels the IP can process. On the other hand, typology parameters contain information about the network typology. These two concepts will be applied to the rest of the thesis.

1.3 Thesis Outline

Chapter 2 concentrates on the background knowledge of Convolutional Neural Network and explains briefly about its typical components. It also gives the reason why YOLOv3-tiny rather than other networks becomes this design's interest. In Chapter 3, popular techniques on how to optimise CNNs on FPGAs are discussed. Afterwards, recent researches about accelerating YOLO on FPGA are shown.

Chapter 4 dives into hardware IP design of each individual layer. For each IP, main methods and algorithms are explained before demonstrating corresponding models. On the other hand, Chapter 5 focuses more on the system-level design and optimisations. A complete model for the network is built for design space exploration. The optimal performance of the design space is compared with other similar works in Chapter 6. Finally, Chapter 7 concludes contributions of this work and discusses future improvement.

Chapter 2

Background

THE focus of this project is mapping a specific neural network, YOLOv3-tiny, on FPGA platforms. In this chapter, background information about neural networks is given to support the choice of YOLO.

Section 2.1 introduces the basic concepts of Convolution Neural Network. Section 2.2 covers the development and trend of network typologies in terms of object detection. Section 2.3 turns to main features of the YOLO typology and explains its advantages in embedded applications.

2.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are bio-inspired, as connections between artificial neurons are similar to biological synapses. In contrast with fully-connected networks, in which every neuron is linked to all neurons in the next layer, CNN has a relatively concise organisation. Because each neuron only responds to a certain receptive field. The reduction on connectivity means lower computational complexity. Therefore, CNN has gained its popularity in terms of image classification, object detection and signal processing.

2.1.1 Convolutional Layer

Inside CNNs, major computation comes from convolution filters. For a typical RGB image input, it has three input channels (feature maps). And each channel can be represented as a 2-D matrix. If a convolutional layer has N_{in} input channels, there will be N_{out} convolution filters which convert inputs into N_{out} output channels, based on (2.1). f_i is the ith input channel, and g_j is the jth output channel.

$$g_j = \sum_{i=1}^{N_{in}} f_i * w_{i,j} + b_j, \quad with \quad j \in [1, N_{out}] \tag{2.1}$$

$w_{i,j}$ represents a convolutional window filter, whose size $k_h \times k_w$ is determined by designers in advance. Values inside $w_{i,j}$ and b_j are called weights and biases respectively. In order to calculate (forward) one convolutional layer, $k_h \times k_w \times N_{in} \times N_{out}$ weights and N_{out} biases are needed in total.

When the height and width of each output feature map are assumed as g_h and g_w respectively, the entire workload of the layer using floating point numbers can be estimated by:

$$workload = g_h \times g_w \times k_h \times k_w \times N_{in} \times N_{out} \times 2 \quad (FLOPs) \tag{2.2}$$

Here, the value is doubled because there are almost equal numbers of accumulations and multiplications. g_h and g_w usually depend on the size of inputs, whether inputs are padded at the edges and the stride of sliding windows.

2.1.2 Other Typical Layer Types

Apart from convolutional layers, many other layer types are also proposed to serve purposes like simulating non-linearity, compressing data, or providing more meaningful outputs.

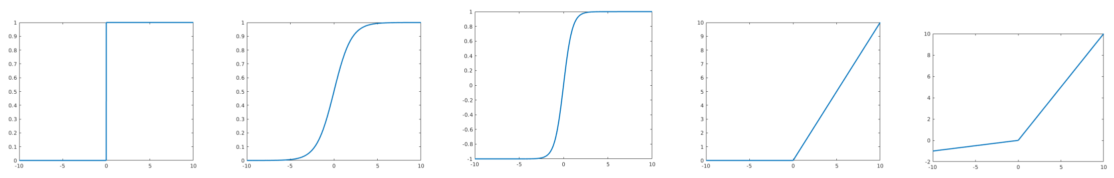
Batch normalisation layer is often introduced to improve stability of the network. Original input data is usually white and not everywhere differentiable [6]. Batch normalisation is actually a technique that transforms data distribution.

$$\hat{g}_j = \frac{g_j - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.3}$$

$$g'_j = \gamma \hat{g}_j + \beta \tag{2.4}$$

g_j and g'_j are inputs and outputs of the layer. μ_B and σ_B^2 are mini-batch mean and variance respectively. ϵ is a tiny value to ensure the denominator is non-zero. γ and β scales and shifts normalised $\hat{g}_j$, and their values are learned during training.

None-linearity layer is widely used to provide necessary non-linearity, since multiplication and accumulation in convolutional layer are purely linear. However, there are many arguments on choosing non-linear functions. Possible candidates include binary step, sigmoid, tanh, ReLU and so on. Currently, it seems ReLU family is very popular among recent designs [7] [8].



(a) binary step (b) sigmoid (c) tanh (d) ReLU (e) leaky ReLU

Figure 2.1: Typical activation functions

Pooling layer is a direct and efficient way to reduce the amount of data. Basically, it picks one value to represent a small region. The typical selection method is finding the average or maximum number within the region.

Fully-connected layer is still in use as the final layer, which captures entire features from previous layers and generates desired classification results. But full-connection also introduces more computation.

Softmax layer becomes attractive when image classification is needed. Originally, classification is based on scores achieved on each class. The meaning of these scores is really vague, and the sum of scores on all classes is not a certain value. The role of softmax is turning scores into probabilities, relying on the transform $P_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.

2.2 Network Typologies for Object Detection

CNNs have got much success in image classification, but there is a huge gap between classification and object detection. For detection, a bounding box is desired to provide location information. In addition, the number of potential objects is unknown.

Regions with CNN features (**R-CNN**) proposed by Ross Girshick et al. [1] puts category-independent region proposals before a CNN. The network then computes features and classifies regions. In other words, a pre-filter is applied before the CNN. The main bottleneck of R-CNN is the number of region proposals. Choosing a large value will result in long detection time, while a small one will make network less accurate.

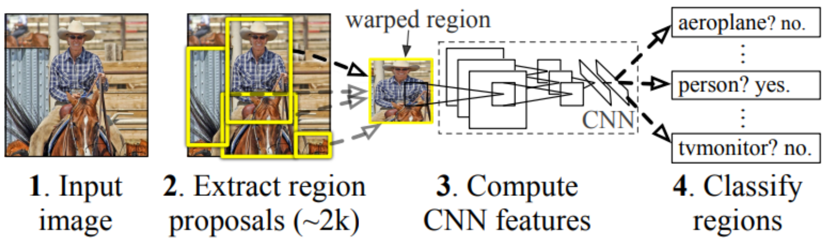


Figure 2.2: R-CNN detection flow [1]

Driven by this, the same author built a faster algorithm called "**Fast R-CNN** [9]. Instead of forwarding the network for each proposal, the original image is directly fed into the CNN. A region of interest (RoI) pooling layer is responsible for extracting features, for individual region proposal. The advantage is evident, as the network runs merely once for each frame.

Shaoqing Ren et al. [10] offered a better solution (**Faster R-CNN**) by replacing region proposal mechanism with a separate neural network. By sharing convolutional layers with the object detection network, the cost of region proposal network is tiny.

Despite such improvement, these mentioned networks still separate object detection into two parts, region proposal and classification. Resource sharing between these parts is limited, which usually results in waste of resource and long detection latency.

2.3 YOLO: Real Time Object Detection

In 2016, Joseph Redmon et al. [2] came up with the first generation You Only Look Once (**YOLO**), which uses a single CNN to predict both the class probabilities and bounding boxes. The main idea is dividing the input image into $S \times S$ grids. Each grid is capable of giving B bounding boxes, with confidence of boxes and probabilities of C classes. The state-of-the-art of YOLO comes from its low latency and high throughput, which are crucial to real time detection.

The improved version, YOLOv2 [11] came in 2017, with techniques such as batch normalisation and anchor boxes. Anchor boxes are predefined bounding boxes with certain scales between width and height. Using anchors instead of a fully-connected layer limits the size of objects, but it decouples object classifications from bounding boxes and improves recall rate.

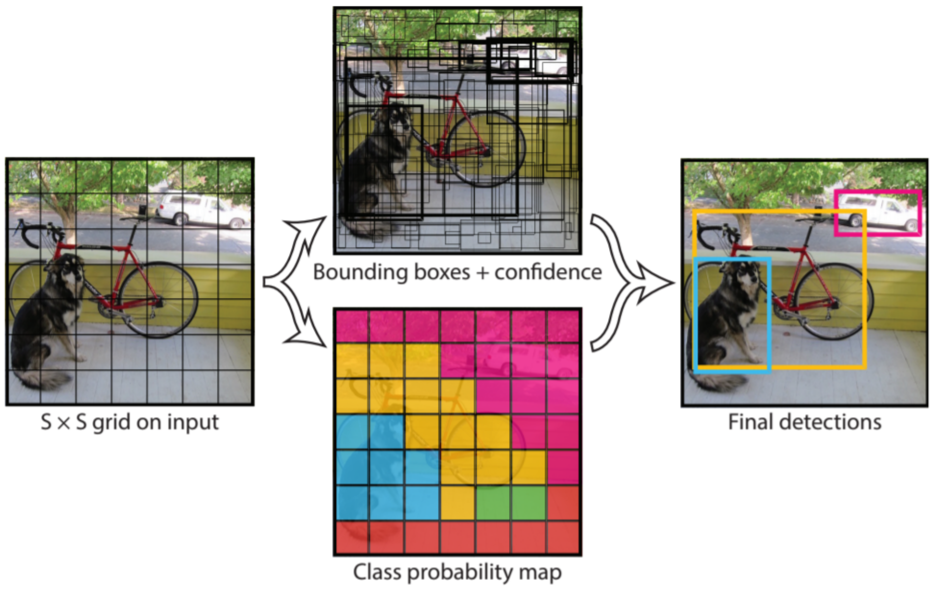


Figure 2.3: YOLO detection example [2]

In 2018, latest YOLOv3 [12] was proposed with higher accuracy. Sigmoid activation is used at the output of network instead of a softmax. Because softmax shows bad performance when multiple labels correspond to the same bounding box. A structure similar to feature pyramid networks is used. The network combines upsampled features with those coming from previous layers. The combined features will be processed further via several convolutional layers. Such structure makes predictions across different scales easier.

For YOLOv3-416 whose workload is 65.86 GFLOPs, it is able to achieve 51.5% mAP50 on COCO test-dev, with 35 FPS on Pascal Titan X GPU. There is also a light-weight model YOLOv3-tiny, which is more suitable for embedded applications and resource-constraint environment. On the same GPU device and test dataset, YOLOv3-tiny can reach 220 FPS and 33.1% mAP50.

Table 2.1: Performance comparison on object detection networks

Model	mAP	FPS
SSD300 [13]	41.2	46
R-FCN [14]	51.9	12
FPN FRCN [15]	59.1	6
Retinanet-50-500 [16]	50.9	14
YOLOv3-416	55.3	35
YOLOv3-tiny	33.1	220

Chapter 3

Literature Review

THIS chapter discusses previous works that deploy Convolutional Neural Networks on FPGAs and special techniques used in embedded applications. The main challenge comes from large computation and memory storage required for CNNs.

3.1 2-D Convolution on FPGA

There are many researches trying to optimise 2-D convolution on embedded FPGA devices. The key point is about memory storage, data bandwidth and parallelism that can achieve.

There are two main buffer schemes, full buffering (FB) and partial buffering (PB) [17]. Full buffering accesses the entire feature map in a line-by-line sequence. It ensures that every input pixel is read from external memories only once. Data will then be stored locally and also reused by sliding window. As a result, this strategy usually demands large on-chip memory on FPGA.

Alternatively, partial buffering divides a feature map into several regions. Each time, only one region will be loaded to the FPGA device. The drawback of a smaller buffer is the same data will be loaded multiple times. As a result, it accesses external memory more frequently and thereby needs higher bandwidth.

A special case of PB is using image to column (Im2Col) algorithm [18] [19]. The input feature map is split into batches which have the same size as weight windows.

The convolution is then transformed into matrix multiplication. Matrices can be further expanded into 1-D vectors. Convolution becomes the dot product of the input vector and the weight vector.

Channels of a standard convolution correlate with each other and have dependency. To make the process more independent, it can be transformed into a depthwise convolution followed by a pointwise convolution. Depthwise convolution performs separable convolutions for each input channel, and they can be parallelised on hardware [20]. Essentially, pointwise convolution is a 1×1 standard convolution that builds cross-channel correlation [21].

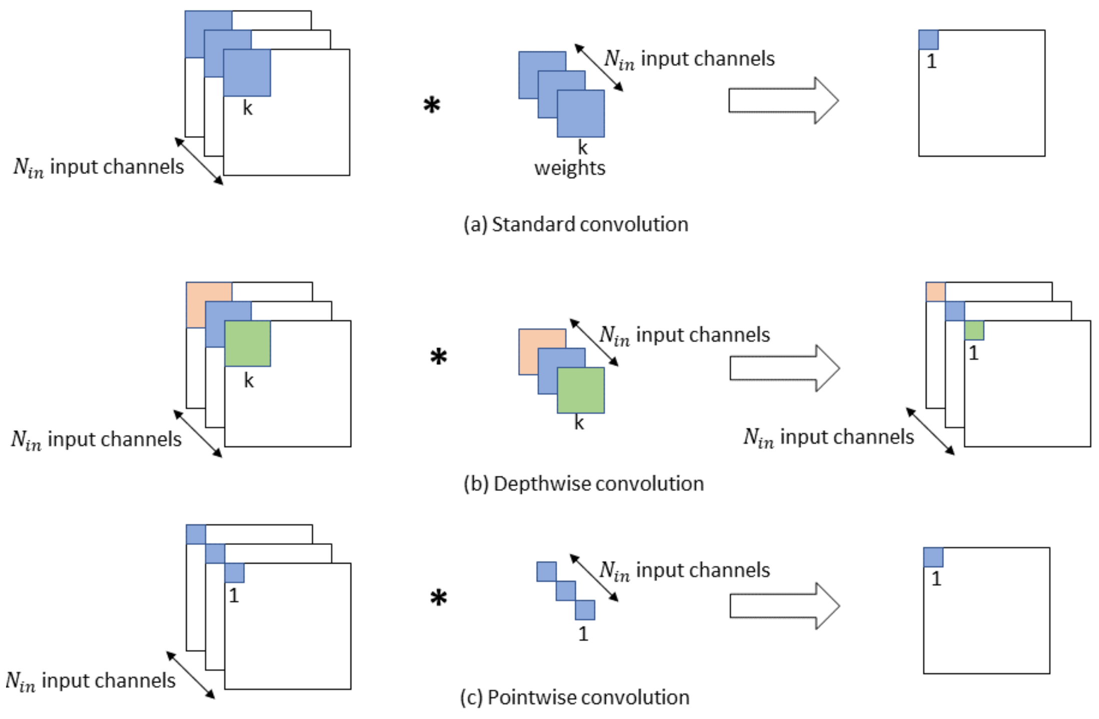


Figure 3.1: Depthwise and pointwise convolution

3.2 Fixed-point Quantisation

As there are no floating-point hard cores on most FPGAs, it is not friendly to keep such high precision. A typical method is training the network on GPU using floating-point representations. The network is then converted to a lower precision version with or without retraining [22] [23].

In fixed-point implementation, it is important to firstly determine how many bits are used for integer and fraction part. For a linear quantisation, bitwidth of integers is related to extremal values and whether overflow will happen. On the other hand, the length of the fraction part affects quantisation error. In addition, the step size of quantisation will shape the distribution of data [23]. For different network typologies, a thorough investigation should be carried out in terms of the trade-off between bitwidth and network precision.

In 2016, Qiu et al. [24] proposed a dynamic strategy that chooses different fraction lengths for individual layer. Re-quantisation will happen between layers, which is implemented by shifting bits in activation. Dynamic quantisation uses resources more efficiently, especially when weight and data distributions vary a lot between layers.

Apart from linear quantisation, there are other methods to further reduce bitwidth. Logarithmic data representation converts weight and activation into log domain. As original distributions are usually not uniform, compression can therefore be achieved [25].

Rastegari et al. [26] proposed Binary-Weight-Networks and XNOR-Networks, which binarised weights only or both data and weights. But most researches on binarised networks focus on image classification rather then object detection. As binarisation will cause huge precision loss, especially on predictions of bounding boxes.

3.3 Dynamic Configurable Architecture

Because layers within a network have different typology parameters. One possible strategy is designing the specific hardware for each layer. It is beneficial to reduce off-chip memory transactions. As each layer is customised, better performance can also be achieved. The disadvantage is that the resource requirement will be quite high by putting the entire network on hardware.

One alternative is separating the design into multiple bit files and hardware needs reprogramming when the network is forwarding. Once a layer is finished, FPGA must be entirely reconfigured. The problem is reloading bit files takes hundreds of milliseconds, which could exceed actual execution time [27]. Although latency of each layer is optimal, the overall performance will suffer from reloading.

A compromised idea is getting a unified hardware architecture that fits all layers. The architecture can work under many settings by controlling multiplexes inside. At present, a lot of CNN implementations on FPGAs tend to adopt this strategy. It is firstly proposed by Chakradhar et al. [3] in 2010, which dynamically configures convolution typology parameters at run-time. Such architecture is also suitable for designs where intermediate results are too huge to be stored on chip. The disadvantage is that a unified framework needs to sacrifice some performance to provide compatibility.

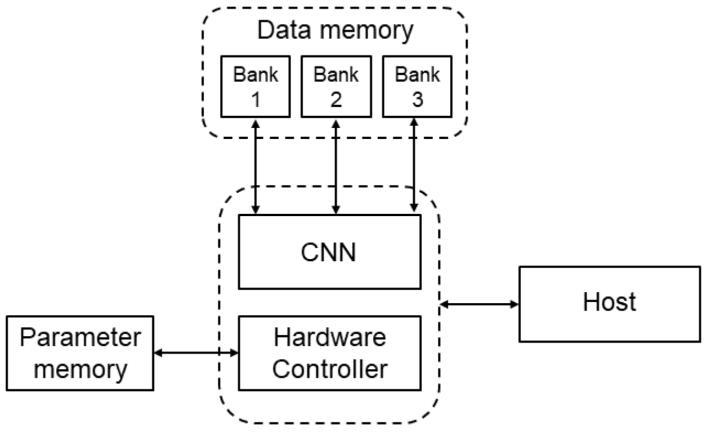


Figure 3.2: A dynamic configurable CNN architecture [3]

3.4 Design Space Exploration

For a tunable design, it is necessary to find out the optimal design point. A direct way is to go over the whole design process for each design parameter choice. The problem is that such search process takes long time for FPGA synthesis, place and route.

Design Space Exploration (DSE) is a method to explore possible design options with models. For each architecture, it corresponds to a design space, within which there are many design points. These design points reflect the relationship between architecture design parameters and performance. Instead of really implementing all design points, a promising solution is using mathematical models for evaluation. It offers a systematic analysis of tunable design parameters before final implementation.

In 2013, M. Peemen [28] designed parameterised HLS templates for CNN, and the space search concentrated on various memory sizes. C. Zhang et. al [29] calculated computation roof and operation intensity, given a combination of loop tile sizes. The aim is to filter out any illegal solution, when the platform limitations are known. N. Suda et. al [30] used design space exploration to choose the optimal performance point that meets resource constraints. Based on performance and resource utilisation models, an optimisation framework is built. H. Sharma et. al [31] described a scalable template for a CNN accelerator, which includes scheduling operations and then binding them to available computing and memory resources.

In 2017, S. Venieris [32] proposed the latency-driven and the throughput-driven flows based on design space models. According to different applications, the design space exploration can focus on a certain optimisation target. A. Rahman [33] proposed a framework able to explore a larger set of design options within CNNs, which includes order of nested loops, various shapes and sizes of MAC array. An accurate model on DSP blocks and latency is finally given.

3.5 Related Work on YOLO

Since 2017, there are many researches on deploying the first and second generation of yolo on FPGA. G. Wei et. al [34] implemented YOLOv2-tiny on a dynamic configurable architecture. Leaky ReLU activation is replaced by ReLU to reduce resources. Parameters of layers are stored in local memory. B. Liu et. al [35] used FPGA as acceleration device for Caffe. Liu combined adjacent layers to reduce communications between host and device. 16-bit fixed point data is used and multiplication products are kept as 32 bits.

Y. Wai et. al [36] proposed a fixed-point version of YOLOv2-tiny, which merged batch normalisation into convolution layers to reduce redundancy. The 3-Dimension

(3D) convolution is implemented as General Matrix-Matrix Multiplication (GeMM) using OpenCL. The design can be tuned by changing multiplication block size and Single Instruction Multiple Data (SIMD) vectorisation factor.

In 2018, H. Nakahara et. al [37] put forward a further lightweight version of YOLOv2. Layers responsible for feature extractions are converted into binary multiply-add operations to improve speed and save power. A parallel support vector machine (SVM) is implemented in floating point numbers.

In 2019, D. Nguyen et. al [38] implemented a high-throughput design by pipelining layers and frames. Stream data is transferred by row between DDR and FPGA. Instead of having a configurable architecture, each layer is mapped to a dedicated hardware block. This minimises unnecessary data exchange and allows multiple image detections to be pipelined for higher throughput.

Chapter 4

Hardware IP Design

4.1 Convolutional Layer

THIS section focuses on the design of the convolutional IP, which is the most important computation element in the system. 4.1.1 gives an overview of the module, including interfaces and major components. 4.1.2 to 4.1.8 explain design details and some optimisations used for FPGA deployment. 4.1.9 introduces HLS directives applied in this work and shows their effects on synthesis. 4.1.10 covers the method to validate behaviour of the IP. Finally, 4.1.11 summarises the convolutional IP by a model, presenting relationships between parameters, latency and resources.

4.1.1 Overview

Show in Figure 4.1, interfaces of convolutional IP contain data ports and control ports. Layer weights, inputs and outputs are transferred through data ports. Considering the amount of data, these ports are implemented as AXI4-stream to provide high speed [39]. Weights and inputs share the same 64-bit port, as weights are transferred before inputs. For those typology parameters, they are set through simple AXI4-Lite connections. Each parameter will be mapped to an independent memory space in PS.

To use the IP, typology parameters should be sent to control logic firstly. Once finished, weights are then transferred from DDR to the corresponding local memory through the input stream. After all preparations are done, layer inputs will flow through the IP and outputs are fed back to DDR. The whole data flow constitutes a loop between FPGA and off-chip memory [40]. The processing system is responsible for managing the whole process.

Inside the module, multiple processing elements should be ready for performing convolutions in parallel, providing hardware acceleration. In addition, inputs and outputs should be buffered. Since data will be reused and the convolution is not a one-to-one mapping.

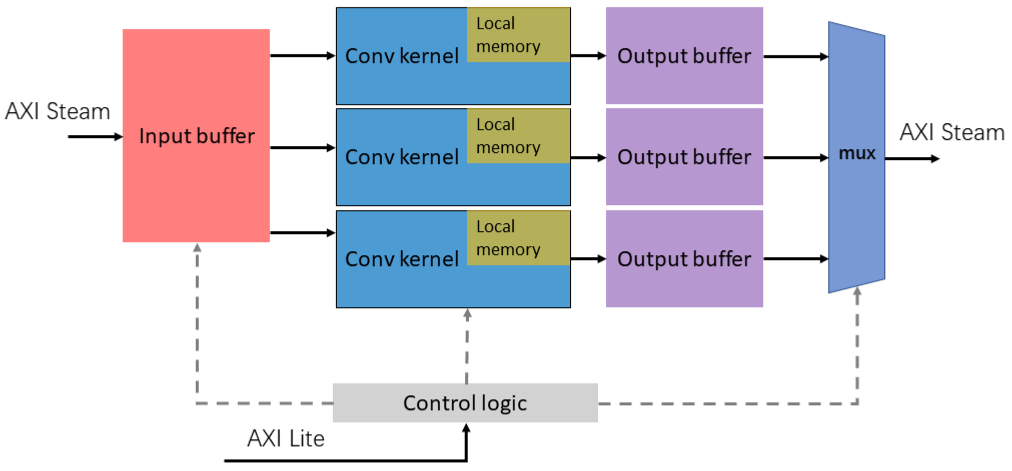


Figure 4.1: Structure overview of the convolutional IP

4.1.2 Merge Batch Normalisation

As mentioned in Chapter 2, most convolutional layers in YOLOv3-tiny are followed by batch normalisation before activation. To perform batch normalisation, based on (2.3) and (2.4), $4N_{out}$ parameters would be transferred and stored on FPGA. It also brings $2 \times g_h \times g_w \times N_{out}$ multiplications and $2 \times g_h \times g_w \times N_{out}$ additions.

However, this FPGA design is only for testing rather than training. Therefore, it is safe to remove batch normalisation with some mathematical techniques. By substituting $\hat{g}_i$ in (2.4) with (2.3), the following equation can be easily obtained.

$$g'_j = \frac{\gamma}{\sqrt{\sigma_B^2+\epsilon}} g_j + \beta - \frac{\gamma\mu_B}{\sqrt{\sigma_B^2+\epsilon}} \tag{4.1}$$

g_j comes from (2.1), where b_j is zero. Therefore,

$$g'_j = \sum_{i=1}^{N_{in}} f_i * (w_{i,j}\frac{\gamma}{\sqrt{\sigma_B^2+\epsilon}}) + \beta - \frac{\gamma\mu_B}{\sqrt{\sigma_B^2+\epsilon}}, \quad with \quad j \in [1, N_{out}] \tag{4.2}$$

If $w_{i,j}\frac{\gamma}{\sqrt{\sigma_B^2+\epsilon}} = w'_{i,j}$, and $\beta - \frac{\gamma\mu_B}{\sqrt{\sigma_B^2+\epsilon}} = b'_j$,

$$g'_j = \sum_{i=1}^{N_{in}} f_i * w'_{i,j} + b'_j, \quad with \quad j \in [1, N_{out}] \tag{4.3}$$

Finally, operations of batch normalisation are merged into convolution [36]. $w'_{i,j}$ and b'_j are new weights and biases. Because such transformation can be finished before run-time, it is almost cost-free except some negligible precision losses during floating point calculations. Merging batch normalisation also shapes the distribution of weights. Figure 4.2 shows the effect on the first convolutional layer in YOLOv3-tiny. As a result, it will inevitably affect design decisions in fixed-point quantisation. This part will be discussed later in more details.

4.1.3 Line Buffer

Chapter 3 mentions the concept of full buffering (FB) and partial buffering (PB). The original YOLO software programme relies on Im2Col algorithm for convolution. Therefore, it is possible to still use PB on hardware. However, Im2Col involves memory movement and data copy on the processor. Evaluated on the ARM processor of Zedboard, Im2Col functions of all layers will take more than 2 seconds. In that case, the system latency will never be lower than 2 seconds, no matter how much hardware acceleration is achieved.

Instead, this design turns to FB method. Although it is called full buffering, FPGA does not need to keep every input all the time. As a window is slid by row-major, only recent rows will be reused in the future. The memory structure utilised here is line buffer,

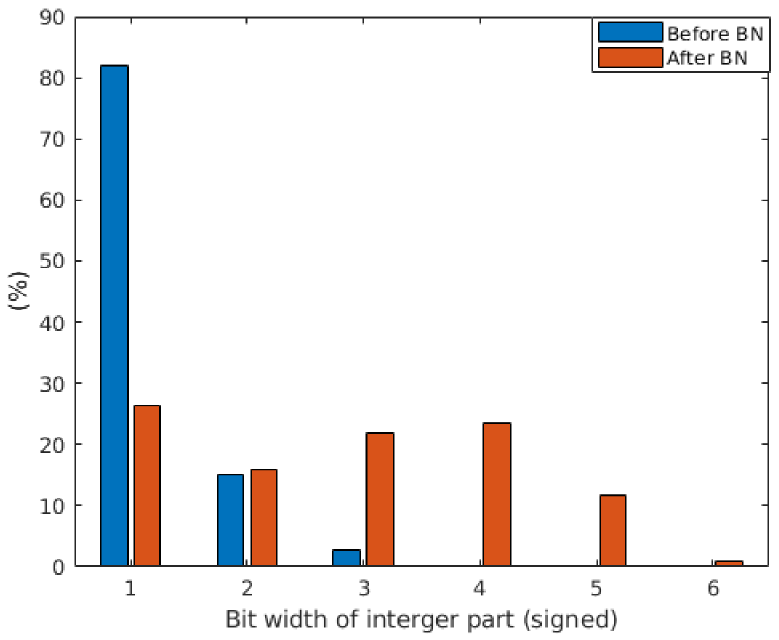


Figure 4.2: Effect of merging BN on weights distribution

which essentially is a shift register array. Suppose the width and height of the input is f_w and f_h respectively. The size of convolutional kernel is $k_w \times k_h$. Then the buffer of one channel has k_h lines, each line containing f_w words.

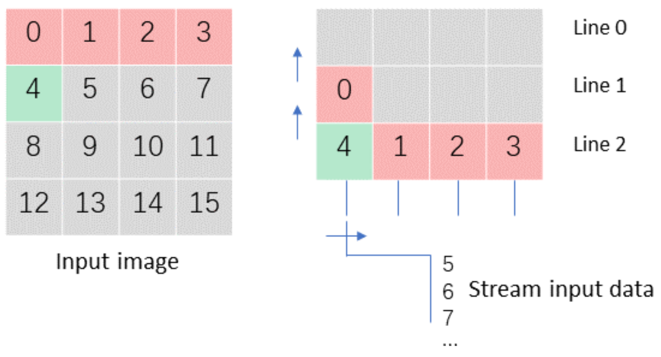


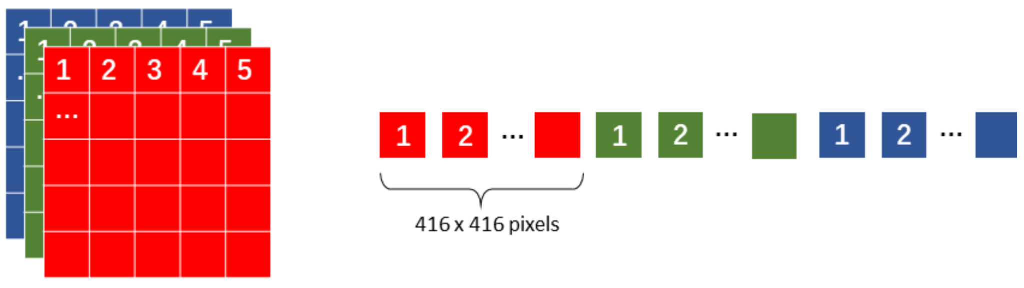
Figure 4.3: Data allocation of line buffer

Whenever an input is captured, data in the corresponding column will be shifted up and the new input will then be inserted (Figure 4.3). As a result, the line buffer will always keep the most recent k_h rows. In YOLOv3-tiny typology, the maximums of k_h and f_w are 3 and 416 respectively. But the line buffer has 3 lines, and 418 words for each row in reality. The extra two columns are caused by convolution padding. If padded zeros are not stored, extra multiplexers will be generated. The decision is trading affordable memory cost for simpler control logic. As long control path is likely to trigger timing violations.

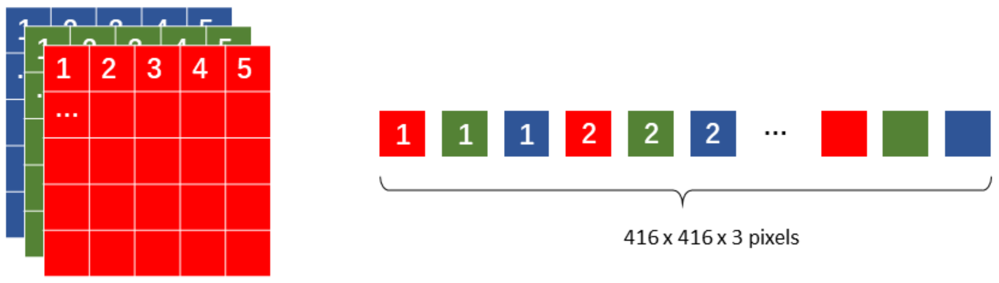
4.1.4 Channel Interleaving

There are three different orders to transfer data between FPGA and off-chip memory. From (2.1), convolution operators should accumulate over all input channels. Therefore, a straightforward way is to sequentially send all pixels within one input channel before moving to the next, illustrated by Figure 4.4 (a).

Take the first convolutional layer which has RGB inputs for example. Both the input width and height of this layer are 416 pixels. Therefore, 416×416 pixels in "R" channel are sent first, followed by 416×416 pixels in "G" and another 416×416 pixels in "B" channel. The problem with this method lies in the accumulation buffer.



(a) sequential channels



(b) channels interleaving

Figure 4.4: Comparison on sequential channels and interleaved channels

For each output channel, it will accumulate convolution results over all input channels. As a result, an output buffer will be used to store temporary values. The size of this buffer is same as layer outputs $g_h \times g_w$. Because the IP is a unified module, the buffer should be designed for the worst case, and that is $416 \times 416 \times word\ length\,(bits)$. Obviously, the output buffer has such a large size that it cannot be put on the test device (Zynq-7020). If the buffer is implemented with off-chip memory, extra delay will be costly.

The solution is interleaving channels, as Figure 4.4 (b) demonstrates. First, one pixel in ”R” channel is sent, followed by the first pixel coming from ”R” and the first pixel from ”B” channel. Afterwards, the second pixel in ”R” channel joins the queue. Such process will continue until the last pixel. Mathematically, it is a transposed version of the original 3D matrix. After interleaving, the accumulation happens for a single pixel rather than the entire channels. Size of output buffers obviously decreases.

However, channel interleaving is not cost-free. For a sequential channel transmission, the line buffer only stores data of one input channel. However, channel interleaving requires separate line buffers for every input channel. The good news is the total buffer size depends on maximal input channels the IP can handle. Such character is desirable as the buffer size is decided by a design parameter.

To summarise, sequential channel transmission relies on a large and fix-sized output buffer, while channel interleaving needs multiple input buffers. The tunable feature of buffer size makes channels interleaving more attractive.

The third choice is combining two mentioned strategies. Each input channel can be divided into many batches. Batches rather than single pixels are interleaved to achieve a balance between previous methods. But the problem of moving and copy data on processor still remains. Considering the inefficient PS of the test platform, it is not adopted in this project.

4.1.5 Sliding Window

Sliding window is responsible for picking up data from buffers and triggering convolutions at the right cycle. If the kernel size is 3×3, convolution will happen for the first time when top two lines and first three columns are filled. Afterwards, for each new row, the convolution will not start until at least three new elements are fetched. Figure 4.5 shows details of the process.

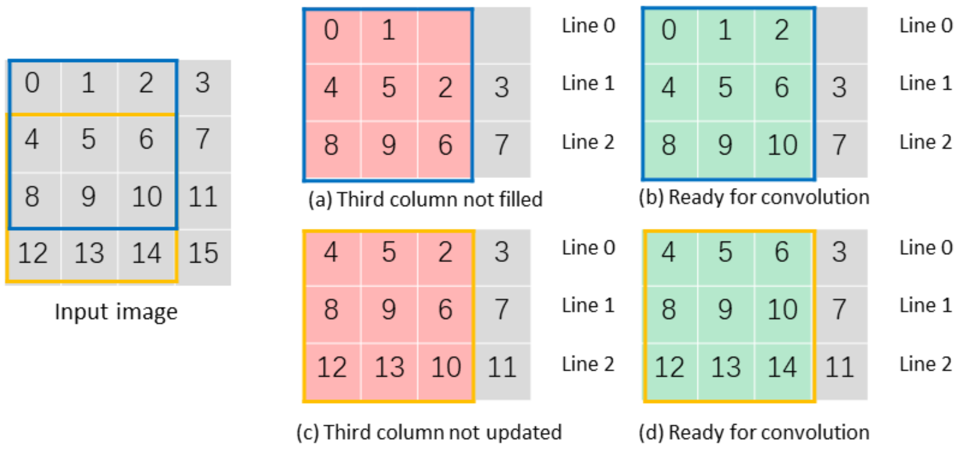


Figure 4.5: Sliding window example for a 3×3 convolution

4.1.6 Multiply-accumulate batch

The main task of the unit is to implement (2.1), and it is divided into two steps. First is the window convolution $f_i * w_{i,j}$, which can also be treated as the inner product of two vectors. For a 3×3 convolution kernel, it includes 9 multiplications and 8 additions. Secondly convolution results of all input channels should be added up, which requires a separate accumulation block.

For a layer with N_{out} output channels and N_{in} input channels, N_{out} convolution units are required and each of them will accumulate for N_{in} times. Parallelism is therefore achieved among different output channels. In this design, inputs belonging to four input channels will be transferred in parallel. Because the bitwidth of data is a quarter of the 64-bit DMA.

It is clearly specified in the code that each multiply-accumulate unit should have four convolution kernels inside to compute four input channels together. In order to save resources, these kernels can share one accumulation unit (Figure 4.6). For clarity, the whole multiply-accumulate unit now is also called "multiply-accumulate batch".

When there are N_{MAC} multiply-accumulate batches, N_{MAC} output channels and four input channels will all run in parallel. In total, $4 \times N_{MAC}$ speed-up is expected, compared with purely sequential execution.

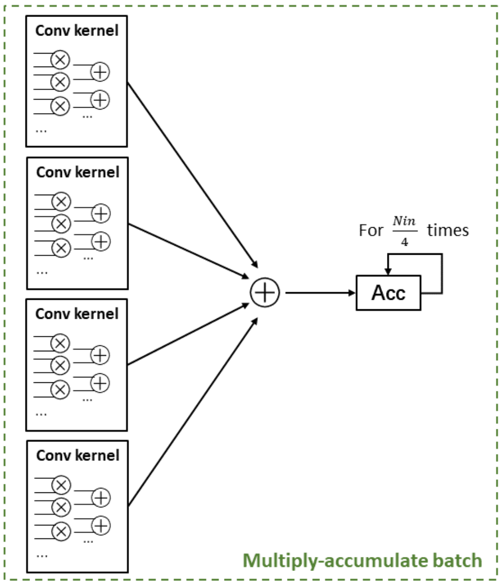


Figure 4.6: A multiply-accumulate batch

4.1.7 Fixed-point Optimisation

Originally, all data and weights in the network are 32-bit floating point numbers. They require large computation capability and memory storage on hardware. One common solution is introducing fixed-point representation, if the accuracy of the system is not affected evidently. By using fixed-point numbers, memory bandwidth can be utilised more efficiently and resources for convolution will be reduced [41].

To implement fixed-point arithmetic efficiently with HLS, Xilinx "ap_fixed" library is used. There are four decisions that should be made, which are total bits, bits of integer parts, quantisation mode and overflow mode [42].

Quantisation mode decides whether rounding or truncation is applied during quantisation. It is discovered that different quantisation modes will not affect resources much. Therefore, truncation is used by default. However, for the overflow mode, Figure 4.7 shows using wrap-around rather than saturation has an evident advantage in LUT and FF.

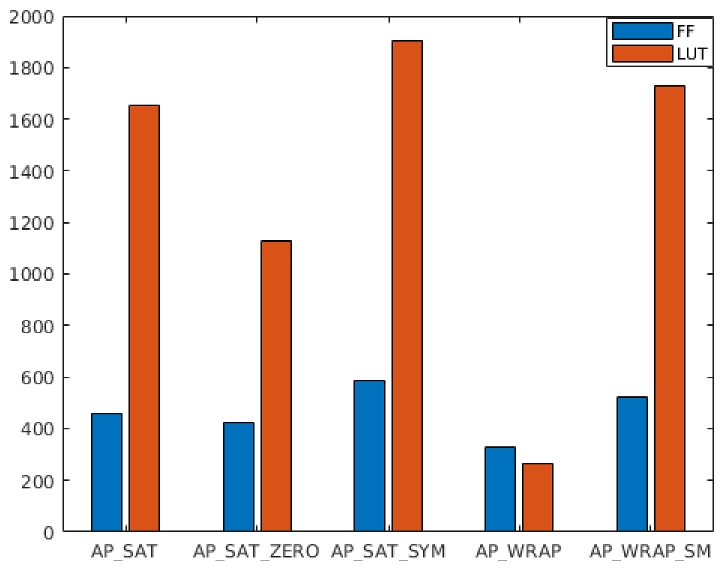


Figure 4.7: Resource comparison on overflow modes of a 3×3 convolution kernel

In terms of total bits, powers of 2 are preferable, as they ease the encoding and decoding process when data transfers happen via 64-bit AXI4-stream protocol. To determine proper bitwidth, experiments are conducted on COCO2014-val5k dataset which contains 5,000 images. Table 4.1 summarises data distribution of all convolutional layers. It is obvious that the absolute values of all inputs and outputs are smaller than 128. It means allocating 8 bits to the signed integer part will be enough without causing any overflow.

Although it seems possible to compress the integer part into 6 or even fewer bits, the actual effect on final accuracy is hard to determine. Running tests for all possible models over 5,000 images could be very time-consuming. For simplification, the attempt is first using 8 bits for integer parts and another 8 bits for fractions. A software fixed-point simulation is built, and achieves 30.9% accuracy, compared with floating version getting 33.4% on COCO2014-val5k. Since such quantisation already affects overall accuracy to some extent, no further compression is carried on here.

It is worth mentioning that the software fixed-point simulation does not behave exactly same as FPGA. Because the "ap_fixed" library for HLS is designed for hardware and is extremely slow for software running. Instead, the simulation just shifts integers to simulate the behaviour of "ap_fixed" library.

Table 4.1: Data distribution of all convolution layers with COCO2014-val5K dataset

Abs	[0,1)	[1,2)	[2,4)	[4,8)	[8,16)	[16,32)	[32,64)	[64,128)	$\geqslant 128$
Input(%)	70.55	17.72	8.14	2.60	0.74	0.24	0.02	~0.00	0.00
Output(%)	65.93	20.03	7.79	3.78	2.22	0.24	0.02	~0.00	0.00

Calculating weight loss can also provide an estimation of the quantisation effect, without running the whole dataset. Loss here is defined by the square law. The weight loss is the ratio of quantisation noise power and signal power [43].

$$weight\ loss = \frac{\sum_{i,j}(w_{i,j} - \hat{w}_{i,j})^2}{\sum_{i,j} w_{i,j}^2} \tag{4.4}$$

$w_{i,j}$ represents original floating point weights. $\hat{w}_{i,j}$ are weights after fixed point quantisation.

Weights distribution is shown in Appendix A.2. The possible candidates of bitwidth are 16 bits (8 integer bits, 8 fraction bits) and 8 bits (4 integer bits, 4 fraction bits). Weights in every layer have different distributions, which implies some of them might be more sensitive to the quantisation while others not. Therefore, weight loss is investigated per layer. Results are shown in the Table 4.2. All weights here have already combined operations of batch normalisation.

Table 4.2: Weight loss of fixed point quantisation

Layer index	8 bit	16 bit
conv1	7.81%	0.00%
conv2	2.68%	0.01%
conv3	3.37%	0.01%
conv4	11.01%	0.05%
conv5	24.26%	0.11%
conv6	48.40%	0.26%
conv7	7.30%	0.03%
conv8	70.91%	0.56%
conv9	17.85%	0.08%
conv10	6.35%	0.03%
conv11	1.36%	0.01%
conv12	60.33%	0.44%
conv13	3.00%	0.01%

Finally, weights are chosen to have the same width as data. Both are represented by 16-bit fixed point numbers. For results of convolution windows, 32 bits are applied to reduce possible overflow and provide higher accuracy for accumulation.

By imposing fixed-point optimisation, data width is compressed from 32 bits to 16 bits, leading to more efficient DMA transfer. In addition, using fixed-point representations brings tangible benefits to latency and resources. Figure 4.8 shows fixed-point implementation saves at least 80% resources, which is very attractive as it makes larger-scale parallelism possible.

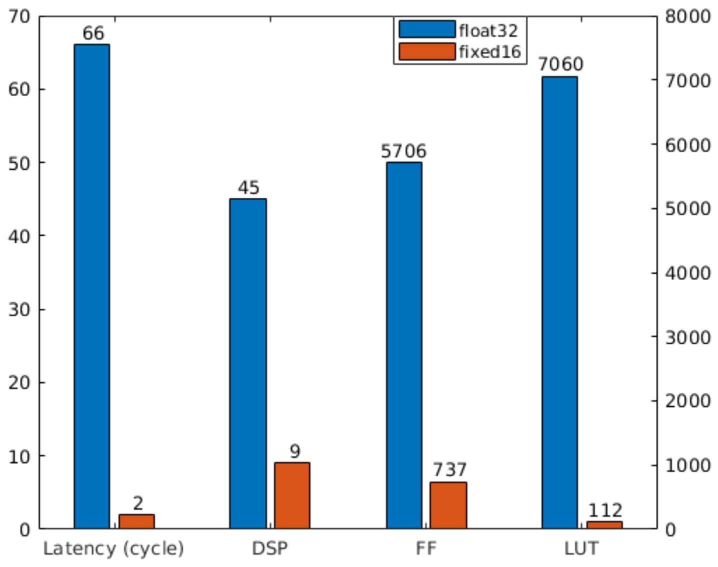
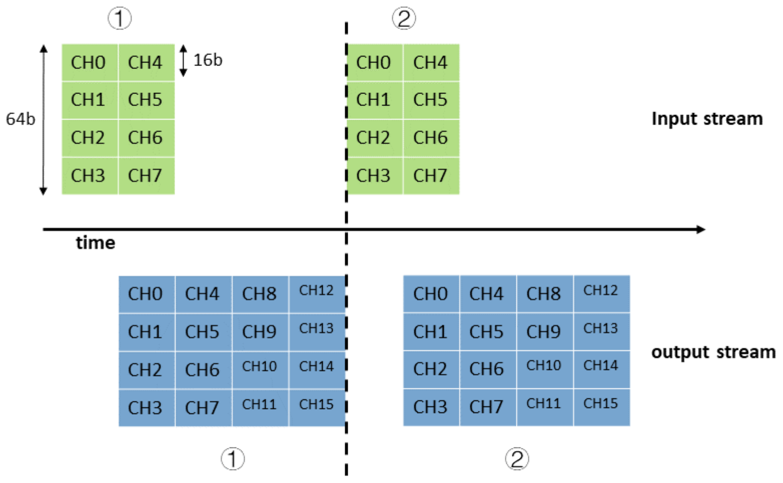


Figure 4.8: Resources and latency comparison on data type of a 3×3 convolution kernel

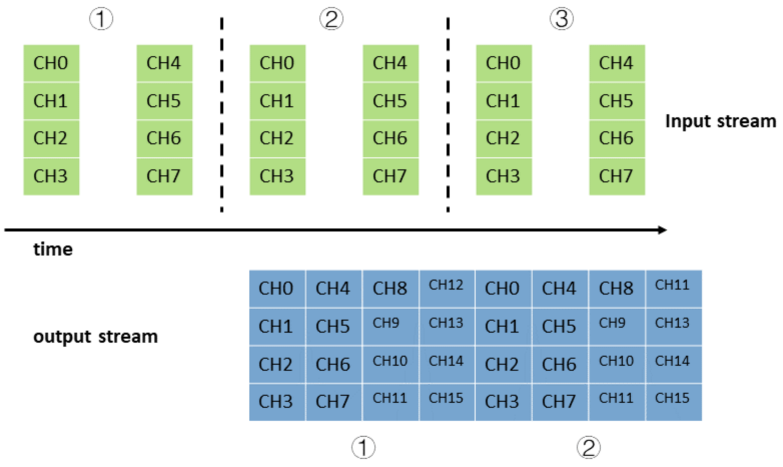
4.1.8 Output Transfer Block

Inside the module, multiple output channels are running in parallel. The convolution needs to accumulate for N_{in} channels, and outputs will be available after the last input channel is processed. In a pipeline, when N_{in} input channels are consumed, N_{out} output channels will be generated simultaneously. In other words, data write operations will happen more frequently than reads, if N_{out} is larger than N_{in}. It will cause data congestion at the output port.

According to the AXI4-stream protocol, only one read and one write transaction is allowed for a port during one clock cycle. As the DMA port is set as 64 bits and each fixed-point data occupies 16 bits, only 4 channels can be read and written. When N_{out} is larger than N_{in}, pipeline has to be stalled until all writes finish.



(a) default stream transfer



①
②
③
CH0 CH1 CH2 CH3
CH4 CH5 CH6 CH7
Input stream
time
output stream
CH0 CH4 CH8 CH12 CH0 CH4 CH8 CH11
CH1 CH5 CH9 CH13 CH1 CH5 CH9 CH13
CH2 CH6 CH10 CH14 CH2 CH6 CH10 CH14
CH3 CH7 CH11 CH15 CH3 CH7 CH11 CH15

(b) improved stream transfer with output FIFOs

Figure 4.9: Stream transfer strategies

A suitable strategy to solve the conflict is adding FIFO between convolution outputs and the AXI4-stream port. The detailed behaviour of the FIFO will be illustrated using the following example. Assume the convolutional layer has 8 input channels and 16 output channels. For a 64-bit stream, it takes 2 cycles to fetch all needed inputs if the initial interval of the module is 1. Assume there is no latency between inputs and outputs. So 16 outputs are waiting for transmission at the second cycle.

As registers storing output values will be cleared when the next 8 input channels come, the input stream has to stall for three extra cycles. Therefore, the whole read and write process takes 5 cycles to complete (Figure 4.9 (a)).

Instead, outputs can be written to FIFOs temporarily. They can be sent gradually when following input data comes, as Figure 4.9 (b) shows. Under this circumstance, the latency is slightly increasing but throughput can be improved. Essentially, the function of output FIFOs is pipelining the stream read and stream write at task level.

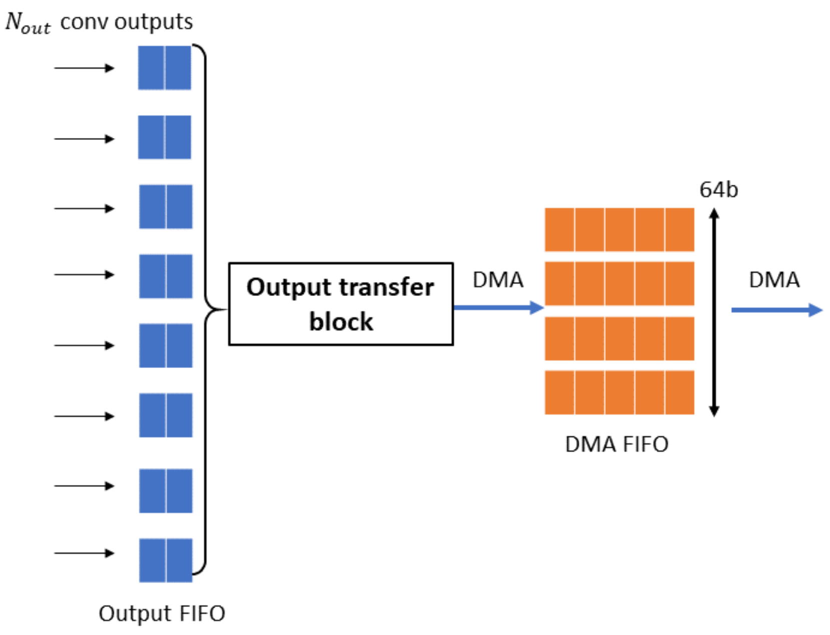


Figure 4.10: Ouput FIFOs and transfer block

4.1.9 Directive Optimisation

•*Pipeline*

Directives in the HLS tool help compiler generate desired hardware. It is important to cover which directives are used and how they affect overall performance. In total, two nested loops constitute this module. The first is responsible for loading weights into the local memory and the second is the main body of convolution operations. As all data are transferred through DMA streams, "PIPELINE" or "DATAFLOW" directives could be applied.

But the accumulation creates dependency between iterations, so "PIPELINE" is more appropriate to be used in this design than "DATAFLOW". The drawback of using a pipeline is all loops inside will be unrolled unconditionally [42]. Therefore, it is hard to tune the performance of individual units and control the number of instances, especially within the multiply-accumulate batch.

Also nested loops can be flattened, when they are perfect or semi-perfect. Perfect loops should satisfy three conditions, only the most inner loop has content, no logic between loop statements and loop bounds are constant. If only the first two requirements are met, it is a semi-perfect loop. To support typology parameters of different layers, loops here have variable bounds and thus are semi-perfect [42].

```
for $i = 0; i < N_{out}; i++$ do
    for $j = 0; j < N_{in}; j++$ do
        for $k = 0; k < 3; k++$ do
            #pragma HLS PIPELINE II=1
            load_weight;
        end
    end
end

for $i = 0; i < f_h + 1; i++$ do
    for $j = 0; j < f_w; j++$ do
        for $k = 0; k < \frac{N_{in}}{4}; k++$ do
            #pragma HLS PIPELINE II=$II_{conv}$
            line_buffer;
            sliding_window;
            conv_mac;
            output_stream_merge;
        end
    end
end
```

Algorithm 1: Pipeline of the convolutional IP

•*Array partition*

The size of weights of a convolutional layer should be $N_{out} \times N_{in} \times k_h \times k_w$. They can be rearranged as a 3-D array, whose size is $(N_{out}, N_{in}, k_h \times k_w)$. The challenge is how to partition the 3-D array and figure out its impact.

The number of blocks in the first dimension corresponds to how many output channels can be in parallel. Local memories are implemented with 2-port BRAMs, thus P_{mem1} blocks in dimension 1 will enable $P_{mem1} \times 2$ convolution kernels or $\frac{P_{mem1}}{2}$multiply-accumulate batches to execute simultaneously. If $\frac{P_{mem1}}{2}$ is smaller than desired output channels N_{out}, some multiply-accumulate batches have to be reused in this pipeline in order to fill the gap. As a result, the pipeline has to be stalled. Memory ports determine the number of multiply-accumulate instances and finally affect total latency.

The second dimension represents different input channels, which can be cyclic partitioned into P_{mem2} blocks. For the 64-bit DMA transfer, four input channels can be parallelised at most. If P_{mem2} exceeds 4, the resources are wasted and no further speed-up is gained.

For the third dimension, the influence goes inside the multiply-accumulate batch. The batch is completely unrolled within the pipeline. For a 3×3 convolution, 9 multiplications should be scheduled in the same cycle if the memory access allows that. Otherwise, unrolled units are not used efficiently and throughput of the batch will decrease. In the following discussion, the third dimension is always completely partitioned.

To summarise this part, the first and second dimension decide how many instances of multiply-accumulate batch are generated. While the third dimension can control performance of the instance itself. All three dimensions can affect overall latency.

4.1.10 Test Bench

In order to validate the convolutional IP, a purely software convolution using "ap_fixed" library is designed. Outputs from the software are compared with those produced by the IP. It is checked through HLS C simulation and C/RTL cosimulation. To finally confirm results, the IP is running on the board and outputs are examined again to ensure hardware behaves exactly same as the software simulation.

4.1.11 Model Analysis

For the convolutional IP, resources and latency can be estimated once design parameters and typology parameters are given. By analysing behaviour of HLS tools, following models can be obtained:

$$\{II_{conv}, BRAM_{conv}, DSP_{conv}\} = M_{conv}(\mathbf{P}) \tag{4.5}$$

$$Latency_{conv} = T_{conv}(II_{conv}, \mathbf{N}) \tag{4.6}$$

$$\mathbf{P} = (P_{mem}, N_{max}, OITR) \tag{4.7}$$

$$\mathbf{N} = (N_{in}, N_{out}, f_h, f_w) \tag{4.8}$$

P is the design parameter vector and **N** is the collection of typology parameters. The model is split into two parts M_{conv} and T_{conv}. The boundary is the process of synthesis, place and route and generating bitstream. M_{conv} reflects the FPGA implementation, and II_{conv}, $BRAM_{conv}$, DSP_{conv} are irrelevant to input data. They are always certain once the bitstream is generated by the tool. On the other hand, $Latency_{conv}$ can change with different settings of the software driver. $Latency_{conv}$ is also dependent on the size of input data.

II_{conv} is the initial interval of the convolutional pipeline. BRAM and DSP represent the number of resources required. P_{mem} is the total block partition factor which is the product of partition factors of dimension 1 (P_{mem1}) and dimension 2 (P_{mem2}). N_{max} is the maximal number of channels the IP can handle. In YOLOv3-tiny, the maximal number of channels of a layer is 1024. However, it is not necessary to keep N_{max} as 1024. Because the network can be reshaped, which is covered in Chapter 5.

OITR is output input transmission ratio defined before. N_{in} and N_{out} are numbers of input channels and output channels respectively in the current layer. Both of them should be smaller than N_{max} after network transformation. f_h and f_w are height and width of the input feature map.

Here the maximum of input channels and that of output channels are forced to be the same value N_{max}. The reason is related to channels interleaving. For instance, there are 16 input channels and 32 output channels in the IP. Through one execution, 32 output channels will be fetched. However, for the next running, data will be re-interleaved to fit 16 input channels.

Previous result on Im2Col already proves that PS of the test device behaves very bad, when dealing with data movement. Therefore, it is reasonable to make the maximum of input channels and that of output channels same in order to avoid re-interleaving.

Currently, no estimation is given for LUT and FF usages. Because it is relatively harder to predict them. HLS tools tend to give a conservative estimation. For this design only, estimation from HLS can double the statistics after place and route. In addition, in the system design, Vivado would explore resources sharing between modules. Other factors may also affect numbers of LUT and FF like timing constraints.

In terms of LUT usages, Figure 4.11 shows the percentage of LUT for multiplexes increases with array partition factor P_{mem1}. Data are gathered when N_{max} is 32, P_{mem2} is equal to 1. In other words, more LUT resources are put into switches rather than real computations. As a result, complex control path prevents further acceleration.

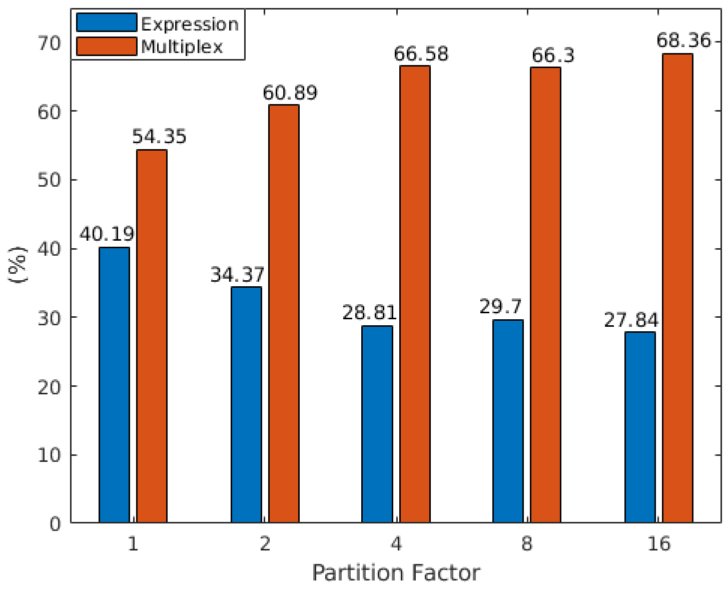


Figure 4.11: LUT usages of the convolutional IP

•M_{conv} : II_{conv} *analysis*

There are three factors that restrict II_{conv}. Based on analysis from 4.1.9, $2 \times P_{mem}$ convolution kernels can be parallelised, but $4 \times N_{max}$ convolutions are waiting for calculations. The pipeline has to stop for $\frac{2 \times N_{max}}{P_{mem}}$ cycles when kernels are reused. $\frac{2 \times N_{max}}{P_{mem}}$ is defined as convolution fold factor F_{conv}.

Secondly, OITR also limits the II_{conv}, because the pipeline has to wait until output writes finish. Since the stream output is wrapped as a function, HLS will force the pipeline to stall for OITR cycles.

Some operations need multiple clock cycles to complete such as reading line buffers. It is then unlikely to schedule the pipeline and make its initial interval equal to one. Actually, these operations set a lowest boundary for II, no matter how many resources are put into the module. Such bottleneck can be removed when algorithm optimisations are proposed. Symbol "OP_{lim}" is used to represent the boundary.

Overall, the II of the convolution pipeline is dominated by:

$$II_{conv} = max(F_{conv}, OITR, OP_{lim}) \tag{4.9}$$

Another way to interpret the equation is that F_{conv} and OP_{lim} are limitations from computation capability, while $OITR$ shows the performance is also memory bounded.

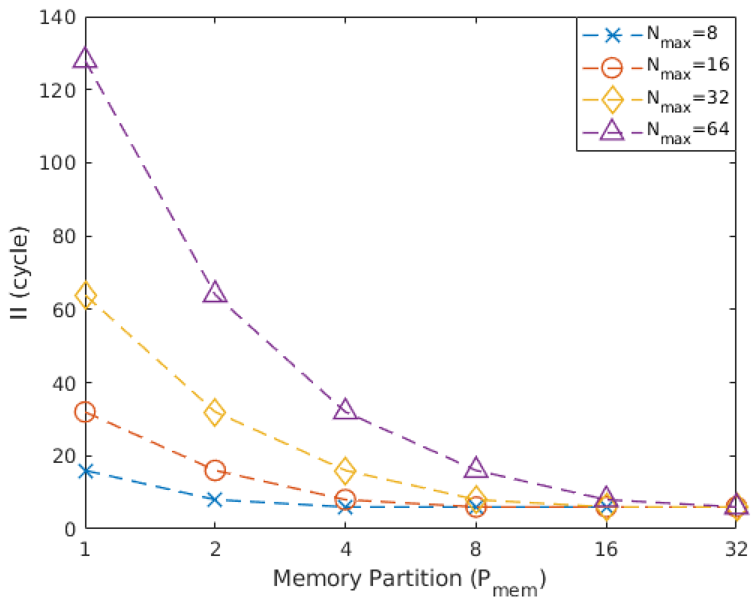


Figure 4.12: Convolutional II due to memory partition

●M_{conv} : $BRAM_{conv}$ *utilisation*

BRAM is mainly used for line buffers and the local weight memory. Size of line buffers is given in section 4.1.3, which is $k_h \times f_w \times N_{max}$ words and each word has 16 bits. Line buffers have been partitioned into 4 blocks to enable parallelism among input channels. Furthermore, HLS tool will automatically implement each line with a separate memory instance.

As a result, line buffers are divided into 12 groups, and each of them contains $418 \times \frac{N_{in}}{4}$ words. Assume all available BRAMs are BRAM18k. Each BRAM can accommodate 1k words for 16-bit data. It is necessary to clarify that BRAM18k in Xilinx devices only supports 1, 2, 4, 9 or 18-bit data access [44]. Although only 16 bits are required, each word will still occupy 18 bits in hardware.

Therefore, it is reasonable to estimate that each line will use $\lceil \frac{N_{max}}{8} \rceil$ BRAMs. For line buffers only, $\lceil \frac{N_{max}}{8} \rceil \times 12$ BRAMs will be used in total.

In terms of local weight memory, it has $N_{out} \times N_{in} \times k_h \times k_w$ words. For YOLOv3-tiny, convolutional kernel size is 1×1 or 3×3, which means k_h and k_w should be set as 3 to fit the extremal case. As the memory is partitioned into $P_{mem} \times 9$ blocks, each block keeps $\frac{N_{out} \times N_{in}}{P_{mem}}$ or $\frac{N_{max} \times N_{max}}{P_{mem}}$ words. Finally, the number of BRAMs for local weight memory can be expressed as $\left\lceil \frac{N_{max} \times N_{max}}{1024 \times P_{mem}} \right\rceil \times P_{mem} \times 9$.

By adding the resource of line buffers and local weight memory together, the utilisation of BRAMs shall be :

$$BRAM_{conv} = \left\lceil \frac{N_{max}}{8} \right\rceil \times 12 + \left\lceil \frac{N_{max}^2}{1024 \times P_{mem}} \right\rceil \times P_{mem} \times 9 \tag{4.10}$$

•M_{conv} : DSP_{conv} *utilisation*

DSP cores are mainly used by the convolution kernel. In the 16-bit fixed point design, each unrolled 3×3kernel needs 9 DSP cores. Therefore, DSP utilisation is linear to the number of kernel instances. From previous discussion, number of instances depends on memory port. But there are other factors which also influence the number.

A possible situation is that F_{conv} is smaller than finally achieved II. In other words, too many instances are created while the throughput of a pipeline is restricted by other reasons. It does not make sense to waste resources without getting speed-up. Under this circumstance, HLS will avoid redundant instances.

During one iteration of the nested loop, $N_{max} \times 4$ convolution operations will be performed. Thus, the number of instances is $\left\lceil \frac{N_{max} \times 4}{II_{conv}} \right\rceil$. One missing part is DSPs used on control logic DSP_{ctrl}, which is a small number but should be included in the current model. In this design, $DSP_{ctrl} = 2$.

$$DSP_{conv} = \left\lceil \frac{N_{max} \times 4}{II_{conv}} \right\rceil \times 9 + DSP_{ctrl} \tag{4.11}$$

•T_{conv} : *Latency analysis*

As explained before, there are two nested loops in the convolutional IP. The first is to load weights of the layer, and the second is for convolution operations. Time spent on two individual loops is defined as T_{weight} and T_{op} respectively. Because each nested loop is pipelined, the latency of can be evaluated by the product of the initial interval and loop tripcount (number of iterations). The depth of the pipeline can be neglected, when it is far smaller than the product.

The initial interval of the first loop can easily reach 1, since the logic behind is simple. It has not been mentioned how to load weights from PS to PL. It takes three cycles to transfer a 3×3 weight and such process will be repeated for $N_{in} \times N_{out}$ times.

$$Latency_{weight} = N_{in} \times N_{out} \times 3 \tag{4.12}$$

The tripcount of the second loop is related to typology parameters f_h,f_w and N_{in}. But the tripcount is not just the product of these three factors. All the convolutional layers in yolov3-tiny are padded, which results in extra two rows and two columns for every input channel. Although these padded zeros will not be transferred, extra clock cycle is desired for sliding windows.

In addition, mentioned in the section 4.1.8, the output is always delayed for one pixel to allow read and write pipeline. Therefore, one more pixel should be added at the end of the loop. The problem is that HLS fails to generate correct hardware, when there are two loops trying to write into one stream. Even though they execute in sequential order and no conflict is guaranteed.

If the extra pixel is put inside the nested loop structure, one entire row has to be added. Because iteration number cannot simply increment by one in a nested structure. In terms of input channels, they are folded by a fixed factor 4 because of the DMA bandwidth.

$$Latency_{op} = (f_h + 3) \times (f_w + 2) \times \left\lceil \frac{N_{in}}{4} \right\rceil \times II_{conv} \tag{4.13}$$

$$Latency_{conv} = Latency_{weight} + Latency_{op} \tag{4.14}$$

4.2 Accumulation & Activation Module

In some cases, N_{max} is smaller than desired N_{in} due to the limitation of resources. Therefore, the convolutional IP cannot accumulate over all input channels through one execution. Instead, the convolutional IP will be launched multiple times for one layer. A separate module is designed whose main task is adding results of all launches together. This method is called ”input channels folding”, and will be explained with more details in Chapter 5.

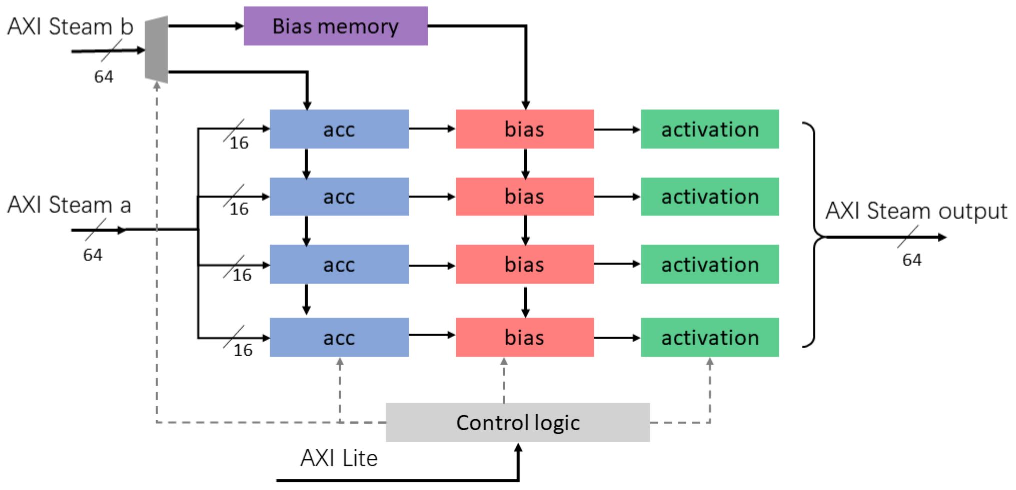


Figure 4.13: Overview of accumulation & activation module

Interfaces of accumulation & activation module include two input streams, one output stream and a bundled AXI4-Lite port for setting parameters. In Figure 4.13, stream a is connected to the output of convolutional IP, while stream b is directly linked with PS to fetch outputs of the previous launch.

After accumulation, the final convolution outputs will be biased and activated. Biases are transferred through the stream b and stored in the local memory, same as weights in the convolutional IP. In YOLOv3-tiny, either linear or leaky ReLU is used for activation. Fixed point multiplication is needed for implementing leaky ReLU.

Similar to convolution, models for accumulation & activation module can be expressed as:

$$\{II_{acc}, BRAM_{acc}, DSP_{acc}\} = M_{acc}(N_{max}, P_{acc}) \tag{4.15}$$

$$Latency_{acc} = T_{acc}(II_{acc}, N_{out}, g_h, g_w) \tag{4.16}$$

P_{acc} is the parallelism factor of this module, which can range from 1 to 4. The potential parallelism exists among different channels, and 4 channels at maximum can be transferred and processed simultaneously because of DMA bandwidth. N_{out}, g_h and g_w represent number of output channels, output height and output width in the corresponding convolutional layer.

$\bullet M_{acc} : II_{acc}$ *analysis*

Again, the module contains two loops for biases loading and accumulation. The initial interval of the second loop depends on:

$$II_{acc} = max(\frac{4}{P_{acc}}, OP_{lim}) \tag{4.17}$$

In this module, OP_{lim} is equal to 1. So II_{acc} is determined by P_{acc} only.

$\bullet M_{acc} : BRAM_{acc}$ *utilisation*

In accumulation & activation module, BRAM is used for local bias memory. Theoretically, it has N_{max} words with 16 bits per word. In YOLOv3-tiny, the maximal number of channels is 1024. It means one BRAM will be enough to store biases, if there is no array partition. However, in order to parallise P_{acc} accumulations, the memory should be $\lceil \frac{P_{acc}}{2} \rceil$ two-port BRAMs.

$$BRAM_{acc} = \left\lceil \frac{P_{acc}}{2} \right\rceil \tag{4.18}$$

$\bullet M_{acc}$: *DSP_{acc} utilisation*

Leaky ReLU functions rely on DSP cores for fixed-point multiplication. Because P_{acc} channels are in parallel, there will be P_{acc} instances of activation function. One more DSP is used in the control path.

$$DSP_{acc} = P_{acc} + DSP_{ctrl} \tag{4.19}$$

$\bullet T_{acc}$: *Latency analysis*

For loading biases, it takes $\frac{N_{out}}{4}$ cycles to finish, since four channels are transmitted together. For accumulation, biasing and activation, the loop tripcount is the product of g_h, g_w and N_{out}. The initial interval is given by (4.17), thus

$$Latency_{bias} = \left\lceil \frac{N_{out}}{4} \right\rceil \tag{4.20}$$

$$Latency_{acc} = Latency_{bias} + II_{acc} \times g_h \times g_w \times \left\lceil \frac{N_{out}}{4} \right\rceil \tag{4.21}$$

4.3 Max Pooling Layer

The internal structure of the max pooling layer is very close to the convolutional layer. Except there is no weight and the filter is max pooling rather than convolution. In the convolutional IP, the concept of stride is not covered. As all strides of convolutional layers are one for this network. But for max pooling, most layers have a stride S_{pool} of two to compress data.

Input height and width of the pooling layer are defined as g_h and g_w, while output height and width are h_h and h_w respectively. In YOLOv3-tiny,the max pooling layer always appears after a convolutional layer. As a result, convolution outputs should have the same size as max pooling inputs.

$$h_h = \frac{g_h}{S_{pool}} \tag{4.22}$$

$$h_w = \frac{g_w}{S_{pool}} \tag{4.23}$$

The padding value for max pooling should be minus infinity rather than zero. Unlike convolution which is padded by a ring, padding for 2×2 pooling will happen when the height or width of inputs is an odd number. The model of max pooing layer can be shown as:

$$\{II_{pool}, BRAM_{pool}, DSP_{pool}, FF_{pool}, LUT_{pool}\} = M_{pool}(N_{max}, P_{pool}) \tag{4.24}$$

$$Latency_{pool} = T_{pool}(II_{pool}, N_{out}, h_h, h_w, S_{pool}) \tag{4.25}$$

$\bullet M_{pool}$: *II_{pool} analysis*

P_{pool} determines how many channels can be in parallel. Same as before, the maximum of P_{pool} is four. Operation limitation here is the access to line buffers, which needs at least two cycles to complete.

$$II_{pool} = max(\frac{4}{P_{pool}}, OP_{lim}) \tag{4.26}$$

$\bullet M_{pool}$: *$BRAM_{pool}$ utilisation*

Line buffers are implemented with BRAMs, and the utilisation is almost same as 4.10 . Kernel size for max pooing is 2×2, so line buffers only have two rows. There are 8 blocks of buffers in total after array partitioning.

$$BRAM_{pool} = \left\lceil \frac{N_{max}}{8} \right\rceil \times 8 \tag{4.27}$$

$\bullet M_{pool}$: *DSP_{pool} utilisation*

As max pooling only involves comparison, no DSP core is required. The remaining source is the control path, and in current design, one extra DSP is utilised.

$$DSP_{pool} = DSP_{ctrl} \tag{4.28}$$

•T_{pool} : *Latency analysis*

Latency of max pooing is straightforward, as there is only one nested loop inside. Therefore, the equation is:

$$Latency_{pool} = II_{pool} \times \left\lceil \frac{h_h}{2} \right\rceil \times 2 \times S_{pool} \times \left\lceil \frac{h_w}{2} \right\rceil \times 2 \times S_{pool} \times \frac{N_{out}}{4} \tag{4.29}$$

Ceiling in the equation corresponds to padding. Number of channels N_{out} is divided by four because of the bandwidth

4.4 Upsample Layer

Upsample layer completes the inverse process of pooling layer. All inputs correspond to the top-left corner of a 2×2 window. When the input data are gathered, it will be firstly stored in a buffer before filling the rest of the window. The buffer only holds the most recent row. It can still be treated as a line buffer but with merely one row. Based on analysis of max pooling layer, following relationships can be easily obtained:

$$II_{upsamp} = max(\frac{4}{P_{upsamp}}, OP_{lim}) \tag{4.30}$$

$$BRAM_{upsamp} = 4 \tag{4.31}$$

$$DSP_{upsamp} = DSP_{ctrl} \tag{4.32}$$

$$Latency_{upsamp} = II_{upsamp} \times g_h \times S_{upsamp} \times g_w \times S_{upsamp} \times \frac{N_{out}}{4} \tag{4.33}$$

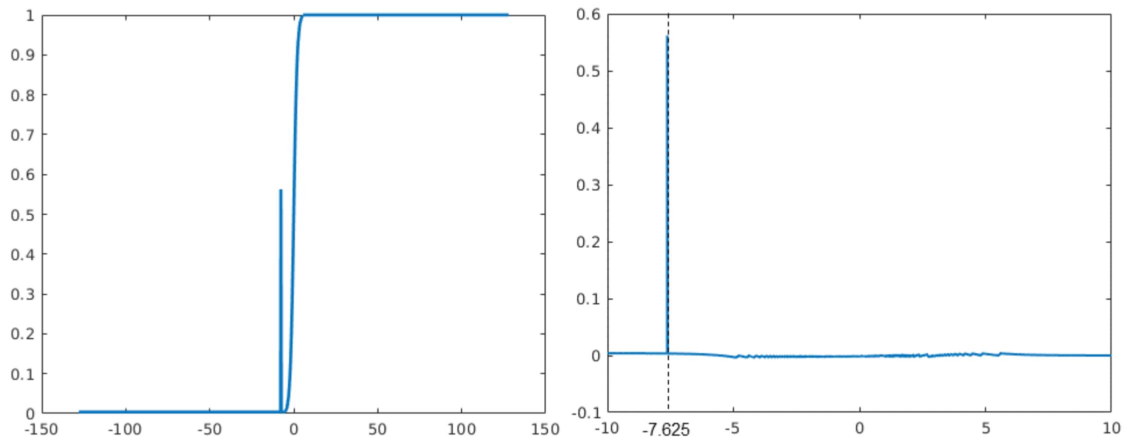
As upsample layer only appears once in YOLOv3-tiny, many typology parameters are certain numbers that will not be exposed as control ports. In fact, both the height (g_h) as well as width (g_w) of layer inputs are 13 and the stride of sliding window (S_{upsamp}) is 2. DSP_{ctrl} in upsample layer is found to be 2.

4.5 Yolo Layer

Yolo layer is tailored for object detection in YOLOv3. For a yolo layer with $g_h \times g_w \times N_{in}$ inputs, it divides the original image into $g_h \times g_w$ grids. Number of channels N_{in} is equal to $(4 + 1 + C) \times B$, where B means how many objects can be detected within one grid, and C the is number of object classes. In YOLOv3-tiny, B is set at 3 and there are 80 classes in COCO dataset.

Therefore, N_{in} of yolo layer has a constant value of 255, which is further divided into 3 groups. In each group, 4 channels provide information on bounding boxes, 1 channel is for objectness score and remaining 80 channels represent individual class scores. Yolo layer applies a sigmoid activation on all channels except those representing the width and height of bounding boxes.

The challenge is to specify which channels should go through the sigmoid function and which should be untouched. The solution is to provide a table, in which each bit implies whether a channel should be transformed. The table is given by the host through simple AXI4-Lite connection, as only 255 bits are waiting for transfer.



(a) fixed-point sigmoid function (b) absolute error

Figure 4.14: Fixed point sigmoid activation

In terms of sigmoid functions, high accuracy is demanded in this design. Because yolo layers are creating final results of the network. The sigmoid activation contains a division and an exponential function. There is a fixed point version of exponential function provided by Xilinx, which provides efficient implementation on hardware [42]. For the fixed point data with 8-bit integer and 8-bit fraction, it will create an approximation shown in Figure 4.14. There is a spike in the implementation, which can be safely removed by adding a switch in hardware. After filtering out the spike, the maximal and average error are 3.9×10^{-3} and 1.9×10^{-3} respectively.

$$II_{yolo} = max(\frac{4}{P_{yolo}}, OP_{lim}) \tag{4.34}$$

$$DSP_{yolo} = P_{yolo} \times 2 \tag{4.35}$$

$$Latency_{yolo} = II_{yolo} \times g_h \times g_w \times \frac{N_{out}}{4} \tag{4.36}$$

OP_{lim} of yolo layer is 1. DSP resources are required for fixed point exponential function, each using 2 DSP cores. The fixed point division is realised by LUT and FF rather than DSP in HLS, though it is not guaranteed after synthesis, place and route in Vivado. Latency is calculated same as before, by multiplying the initial interval and loop tripcount.

Chapter 5

System Design

AS individual IP blocks are ready, this chapter focuses on assembling these IPs into a complete system. A system-level description is firstly provided. Afterwards, an introduction on how to transform the YOLOv3-tiny network is given. The aim is to reshape the network structure for FPGA acceleration. Block-level design and software drivers are introduced afterwards. Based on model analysis before, a system model is finally present to allow Design Space Exploration.

5.1 System Overview

Figure 5.1 demonstrates the system structure. IPs constitute a unified accelerator for YOLOv3-tiny which will be discussed with details in Section 5.3. To transfer large amount of data, off-chip DDR is connected with the CNN Accelerator through AXI4-streams. DMA0 has separate read and write data ports, each with 64 bits. DMA1 only moves data from main memory to slave (MM2S), which is used for accumulation inside the accelerator.

The processor is responsible for coordinating the system through the bundled AXI4-Lite. The connection between PS and the accelerator is also used for sending typology parameters. Some other modules are not shown in the figure, including Processor System Reset and AXI Interconnect. The complete system diagram is attached in Appendix B.1.

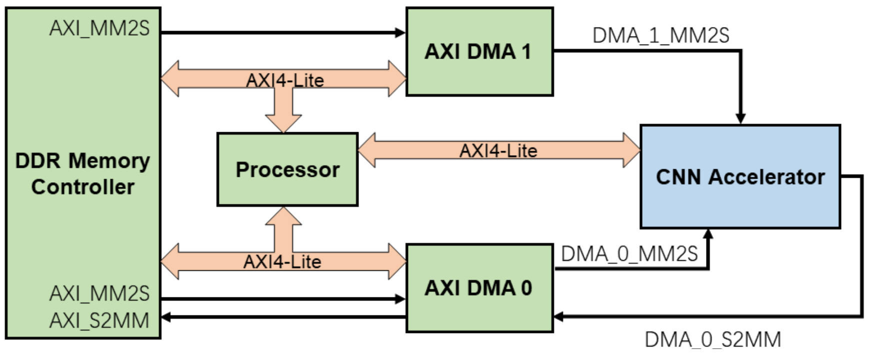


Figure 5.1: System-level connections

5.2 Network Transformation

Although this design is dynamic configurable, the final hardware may not be able to support the original typology parameters because of resources and memory bandwidth limitations. In that case, the typology must be transformed to fit the hardware available. Transformation changes network structure but will not affect final outputs.

5.2.1 Layers Combination

There is a stream loop between off-chip DDR and programmable logic on FPGA. A simple strategy is each layer constitutes a loop. In other words, a layer gets inputs from DDR and sends outputs directly back to DDR. The alternative is wrapping several layers into a group, so the off-chip DMA transfer only happens at the beginning and the end of the group. Fewer off-chip transactions are helpful to reduce latency and power [45].

The idea is using convolutional layers as boundaries to divide the networks, as Figure 5.2 shows. To avoid confusion, groups here are named as ”layer groups”. In YOLOv3-tiny, there are four possible structures of a layer group, including a single convolutional layer, convolution followed by max pooling, convolution followed by yolo or convolution followed by upsampling.

It is also possible that several layer groups can be further merged into a larger unit for hardware running. Each IP will have multiple copies on hardware. In the extremal case, all layers have their specific hardware instances instead of sharing a dynamic configurable architecture.

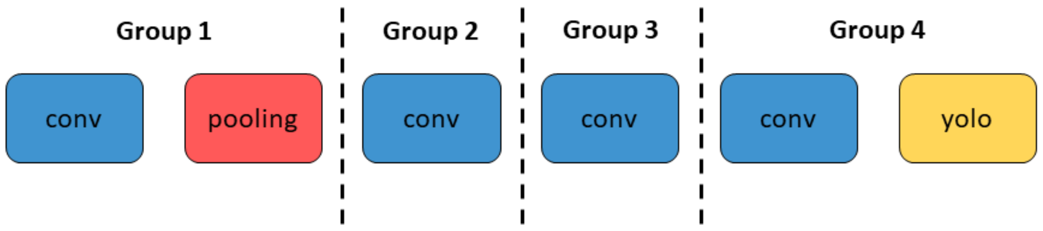


Figure 5.2: Example of layers combination

5.2.2 Channels Folding

The upper limit of channels number N_{max} is restricted by resources available. Therefore, it is a common situation that N_{max} might be smaller than desired N_{in} or N_{out} in the convolutional layer. The solution is to split the original layer into some sub-layers. These sub-layers have fewer channels and they can fit into IPs. This method is called channels folding. When the convolution layer is folded, all other types of layers will follow to avoid re-interleaving channels.

•*input channels folding*

For a convolution layer with N_{in} input channels, assuming N_{out} is smaller than N_{max}, there will be $\left\lceil \frac{N_{in}}{N_{max}} \right\rceil$ sub-layers. $\left\lceil \frac{N_{in}}{N_{max}} \right\rceil$ is also defined as input channels folding factor F_{in}. If N_{in} is divisible by N_{max}, each sub-layer contains N_{max} input channels and N_{out} output channels. The outputs of these sub-layers are $g_{j,1}, g_{j,2}, ..., g_{j,F_{in}}$. Accumulation & activation module is applied on these results to get final outputs g_j. Mathematically, such process can also be explained by splitting (2.1):

$$g_j = \sum_{i=1}^{N_{max}} f_i * w_{i,j} + \sum_{i=N_{max}+1}^{2N_{max}} f_i * w_{i,j} + ... + \sum_{i=(F_{in}-1)N_{max}+1}^{N_{in}} f_i * w_{i,j} + b_j, \quad with \quad j \in [1, N_{out}] \tag{5.1}$$

In other words, input channels folding breaks the accumulation chain into smaller clusters. Equation (4.13) shows latency of convolutional IP has a linear relationship with N_{in}. Theoretically, total latency of F_{in} sub-layers should be equivalent to the original layer.

However, it is not the real case, as there is an extra cost on launching the IP for multiple times.

●*output channels folding*

In contrast, output channels folding happens when N_{out} is larger than N_{max}. Unlike input folding which will not affect total latency much, output channels folding factor will contribute to the latency linearly.

In convolution, each output channel is related to all input channels. When output channels are folded, all inputs have to be sent again. Because data is a stream and only part of inputs are buffered locally. The overall latency of one layer will increase by a factor of $\left\lceil \frac{N_{out}}{N_{max}} \right\rceil$ (or F_{out}). Moreover, extra cost on starting IPs still exists. Figure 5.3 demonstrates an example on channels folding.

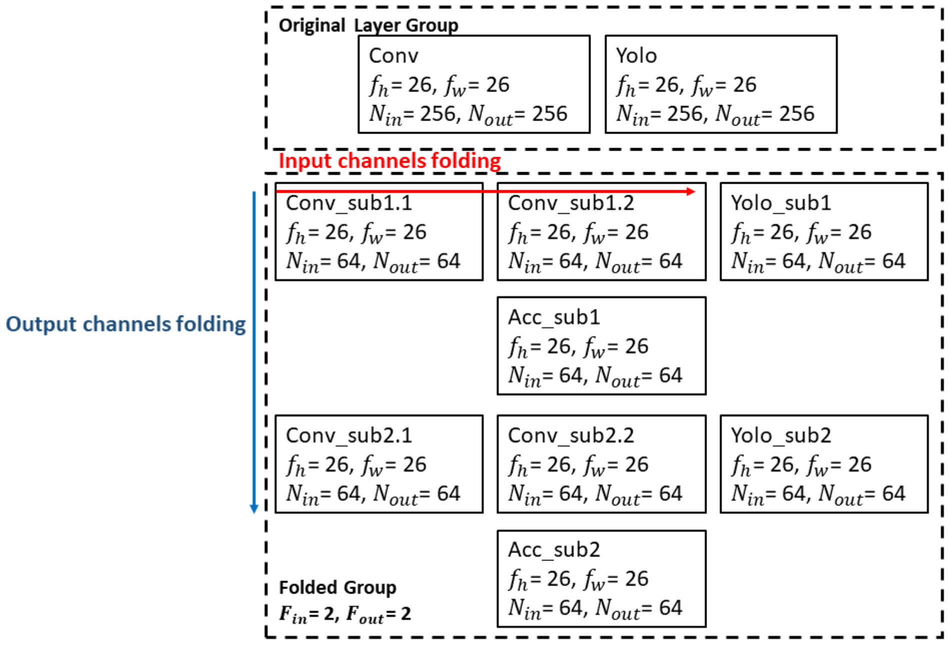


Figure 5.3: Example of channels folding

5.2.3 Channels Padding

As one DMA port transfers four channels in parallel, it is desirable that channels of a layer can be divisible by 4. It applies for most cases except for the RGB inputs which obviously have only three channels, and yolo layers having 255 channels for COCO dataset. As only 5 out of total 24 layers will be affected, they are simply padded by one more channel to satisfy the requirement.

Another situation for channels padding is related to "OITR", mentioned in 4.1.8. OITR affects the initial interval of pipeline, and the value of OITR is determined by the worst case in the network. In YOLOv3-tiny, it is the first layer, whose N_{in} and N_{out} are 3 and 16 respectively.

It is possible to add idle input channels to reduce the ratio. For those padded layers, their performance might drop because of redundancy. However, other layers will benefit from a lower II. The purpose of this design is accelerating the whole network rather than individual layers. Channels padding is a possible strategy from the system level.

5.2.4 Convolution Size Padding

In YOLOv3-tiny, the convolution kernel size is either 3×3 or 1×1. As the convolution is already unrolled, it is hard to share resources between two different kernel sizes. In other words, a 3×3 kernel cannot be reconfigured as 9 1×1 kernels using same resources.

To support 1×1 convolution, a dedicated module will be required, or 1×1 can be padded into a 3×3 convolution. In this design, the latter method is taken. After transforming the kernel size, the inputs are unchanged but 1×1weights will be surrounded by zeros to become a 3×3 window. These idle zeros in weights will cause extra 10 ms in total latency. Although these idle zeros take part in the computation, they are not taken into account for the system workload.

5.3 CNN Accelerator

After transforming the network, IPs designed in Chapter 4 can be categorised into 3 types. First, the convolutional IP fetches weights and input data through DMA, and it feeds data into the accumulation & activation module. Secondly, the accumulation will happen when input channels are folded before biasing and activating. Finally, accumulated and activated outputs might be sent to max pooling, yolo or upsample for further processing depending on specific typology.

Consequently, in Vivado block design, the system contains three processing stages on PL. Stages are connected with AXI4-stream and switch IPs providing configurable routing between masters and slaves [46]. PS and PL communicates through two 64-bit DMA ports and an AXI4-Lite interface.

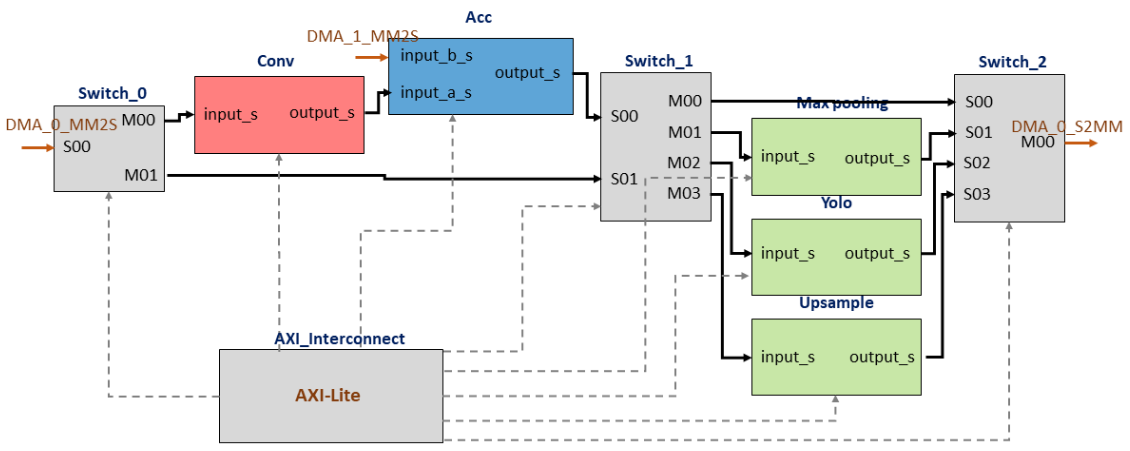


Figure 5.4: Block design of hardware accelerator

Switch 0 and 1 control whether DMA0 inputs flow into the convolutional IP or go to the third stage directly. Although it is efficient to combine the convolution layers and other types of layers together. Binding layers is not always a good idea. For example, when outputs of a convolutional layer will be useful somewhere else in the network. Then results must be captured by the off-chip memory. In such case, even though the convolutional layer is followed by a pooling layer, these two layers have to run separately. That is why the convolutional IP may be bypassed in certain situations.

Switch 1 and switch 2 determine what kind of process should be done to accumulation outputs. It can be max pooling, yolo, upsample or nothing at all. When there is no further processing, it means either the next layer is convolution again or the accumulation of channels folding has not finished yet.

5.4 Processing System Design

5.4.1 Processing Flow

Drivers on processing system cope with three tasks, initialising all peripherals, determining layer groups based on typology and forwarding all groups to get results. The first two parts belong to system initialisation and should only be executed once.

The forwarding part can be further divided into five steps, setting IP parameters, setting routes through switches, starting IPs, sending input stream data and fetching outputs. To detect many images or even deal with a video stream, the forwarding part should run repeatedly.

```
for $i = 0; i < F_{out}; i++$ do
    for $j = 0; j < F_{in}; j++$ do
        set IPs;
        start IPs;
        if Layer group contains Conv then
            send weights;
            if $j == F_{in} - 1$ then
                send biases;
            end
        end
        DMA transfer;
        wait until all IPs are done;
        swap pointers of accumulation buffers;
    end
end
```

Algorithm 2: Software driver for a layer group

In addition, the main body of the forwarding part is a nested loop. The outer loop iterates over output channels folding and the inner one covers input channels folding. When dealing with input channels folding, buffers are needed on DDR to store accumulation inputs and outputs.

For accumulation, it is not a good idea copying data from output buffer to input buffer for the next iteration. In fact, using the ARM processor on Zedboard to copy accumulation data will take up to several hundreds of milliseconds. The alternative is simply swapping pointers in software. It works when input buffer and output buffer have the same size. This further supports the idea that the maximal number of input channels and output channels of IPs should be equal.

5.4.2 Convolutional Weights

Currently, there is no specific 1×1 convolution kernel available in hardware. Therefore, weights of those layers must be padded to 3×3 convolutions. Also, weights are transferred in the unit of kernel window. The size of 3×3 kernel is not divisible by 4. It is possible to build a data pack and fully utilise the 64-bit bandwidth. Therefore, it takes 2.25 cycles per window on average. But it requires extra encoding and decoding circuits. Instead, the kernel size is padded to 12 and it takes 3 entire cycles to transfer each 3×3 window. Choosing this strategy will lead to a constant cost, about 11 ms.

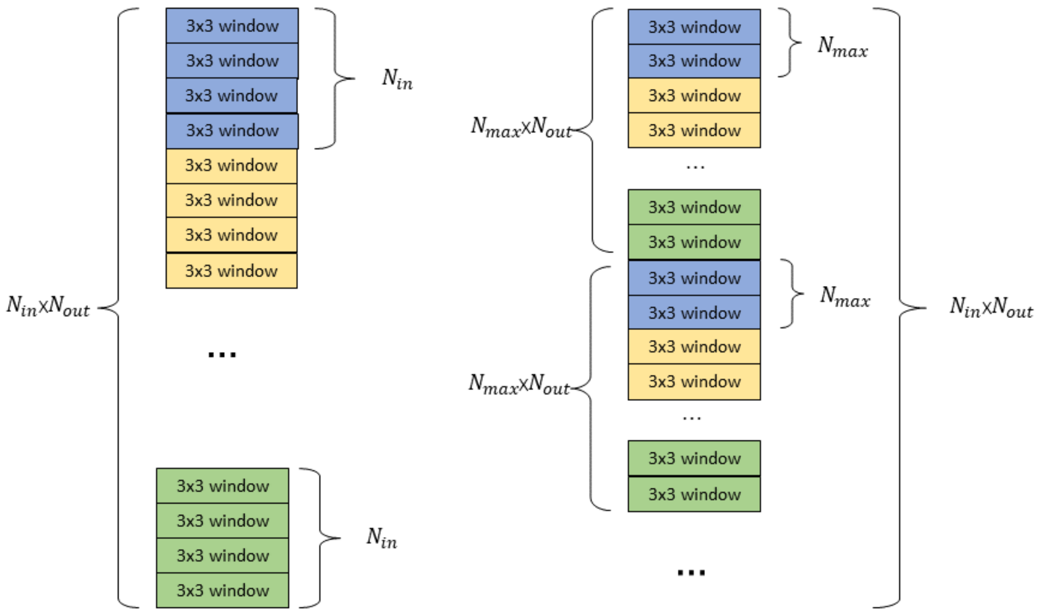


Figure 5.5: Weights rearrangement for channels folding

In order to support channels folding, weights of convolutional layer should be rearranged in advance. Originally, weights are stored in a 3-D array, whose three dimensions correspond to output channels, input channels and kernel windows. The problem is only N_{max} out of N_{in} input channels will be accessed during one DMA transfer, but they are not stored in a continuous space. Therefore, as Figure 5.5 shows, input channels must be reorganised before run-time.

5.4.3 Memory Access

It is significant to figure out total memory transactions for one layer group. Suppose f_h, f_w and N_{in} are input height, input width and number of input channels for the layer group. h_h, h_w and N_{out} are output height, output width and number of output channels for the group respectively.

Total input size of the group $Size_{in}$ is equal to $f_h \times f_w \times \left\lceil \frac{N_{in}}{4} \right\rceil \times 4$, while output size $Size_{out}$ is $h_h \times h_w \times \left\lceil \frac{N_{out}}{4} \right\rceil \times 4$. For accumulation, extra input data transfer size $Size_{acc}$ is $f_h \times f_w \times \left\lceil \frac{N_{out}}{4} \right\rceil \times 4$. Another part comes from weights and biases, which can be expressed by $Size_{weight} = \left\lceil \frac{k_h \times k_w}{4} \right\rceil \times 4 \times N_{in} \times N_{out}$ and $Size_{bias} = \left\lceil \frac{N_{out}}{4} \right\rceil \times 4$ respectively.

To summarise, for layer groups containing convolution, the entire number of memory accesses between off-chip DDR and programmable logic is:

$$Size_{mem} = Size_{in} + Size_{out} + Size_{acc} + Size_{weight} + Size_{bias} \ (words) \qquad (5.2)$$

5.4.4 Image Input

A missing part that has not been covered is the image input of the system. Both the width and height of the network input are 416 pixels. Any images with different sizes must be resized and letterboxed first. All pixel values are then normalised between 0 and 1. Channels interleaving and padding are also implemented. In this design, the focus is the YOLO network itself. Therefore, pre-process on the image is completed before run-time. The image is supposed to be available in off-chip memory before network detection.

5.4.5 Route Layer

Route layer is designed on software rather than FPGA hardware, as it only involves data copy and movement. The function of route layer is redirecting data flow within the network. It is crucial for detections on multiple scales. When route layer has only one layer input, it is realised by simply passing the pointer. When there are multiple input layers, data will be copied into a continuous memory space. The equivalent effect is concatenating data of input layers on the channel dimension.

5.5 Overall Model

Chapter 4 gives models of individual IPs on resources and latency. In terms of resources, the numbers can be simply added up to get an estimation of system resource utilisation. However, overall latency is not the sum of all modules because of pipeline.

5.5.1 Resource Estimation

•DSP_{sys} *utilisation*

DSP utilisation is the sum of all sub-modules, which is

$$DSP_{sys} = DSP_{conv} + DSP_{acc} + DSP_{pool} + DSP_{upsamp} + DSP_{yolo} \tag{5.3}$$

A reasonable approximation is neglecting those for control path, as they can be implemented with LUT and flip-flop instead. The synthesis tool will make the decision based on free resources. Therefore, DSP_{conv} for convolution and DSP_{yolo} for sigmoid activation should be the major concern.

•$BRAM_{sys}$ *utilisation*

System BRAM utilisation is not exactly the sum of all IPs. There are extra FIFOs between PS and PL, which are part of AXI4-stream interfaces. As DMA_0 has both read and write port, but DMA_1 only reads data from DDR. FIFO size of DMA_0 should double that of DMA_1.

$$\begin{aligned} BRAM_{sys} = {} & BRAM_{conv} + BRAM_{acc} + BRAM_{pool} + BRAM_{upsamp} \\ & + BRAM_{yolo} + BRAM_{DMA\_0} + BRAM_{DMA\_1} \end{aligned} \tag{5.4}$$

5.5.2 Hardware Latency Estimation

•II_{sys} *analysis*

As all modules are pipelined and they are connected with each other through stream FIFOs, the whole system is still a pipeline. The initial interval of the system is determined by the module which shows the worst performance.

$$II_{sys} = max(II_{conv}, II_{acc}, II_{pool}, II_{upsamp}, II_{yolo}) \tag{5.5}$$

Since the convolutional IP uses most resources, high II_{conv} is harder to achieve. II_{sys} is mainly dominated by the performance of convolution. Under this circumstance, spending resources on accelerating other modules is not a wise choice. It provides a general guidance on choosing parallelism factors of other modules. In addition, it shrinks the range of design space that deserves exploration.

•*Latency analysis*

The network has been divided into layer groups. Depending on the channels folding factor, a layer group may run hardware IPs for multiple times. One execution of the hardware system is named as one "hardware calling". During one hardware calling, total time includes loading weights to the convolutional IP, loading biases to accumulation module and dataflow through the whole pipeline. Latency for each hardware calling can be approximated by:

$$Latency_{hw\_exe} = (f_h + 3) \times (f_w + 2) \times \left\lceil \frac{N_{in}}{4} \right\rceil \times II_{sys} \tag{5.6}$$

$$Latency_{hw\_call} = Latency_{hw\_exe} + Latency_{weight} + Latency_{bias} \tag{5.7}$$

Equation (5.6) is derived from (4.13). $Latency_{hw\_call}$ only takes hardware delay into account. Delays between PL and PS such as setting parameters through AXI4-Lite are not included here.

For a layer group, hardware will be called for $F_{in} \times F_{out}$ times. Thus, the latency of a group is:

$$Latency_{layer\_group} = F_{out} \times F_{in} \times Latency_{hw\_call} \tag{5.8}$$

The overall hardware latency of forwarding the network can be expressed by:

$$Latency_{network\_hw} = \sum Latency_{layer\_group} \tag{5.9}$$

5.5.3 Software Latency Estimation

The previous section focuses on latency of FPGA hardware only. Extra software execution time on the ARM processor should not be neglected. The most frequent and time-consuming software operations are related to cache and DMA transfer.

The memory region waiting for DMA sending must be flushed away from the data cache and written back to main memory. It ensures data stored in DDR is up to date. After receiving data through DMA, the processor must invalidate the corresponding cache region. It guarantees data will be read from main memory directly and what left in cache will be abandoned. On the test platform, it is discovered that time on flushing or invalidating is almost linear to the DMA transfer size (Figure 5.6).

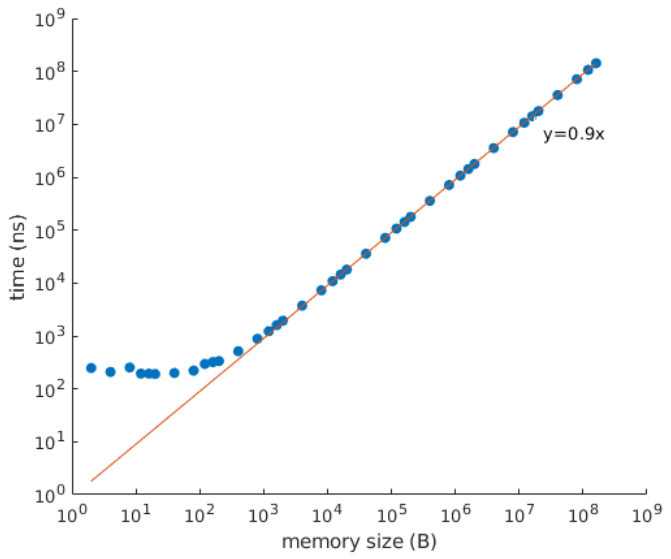


Figure 5.6: Latency experiments on cache flush and invalidation

The size of DMA memory access is given in section 5.3.3. For a layer group containing convolution, the total size of flushing and invalidating operations is:

$$Size_{sw\_op} = 12 \times \overline{f_{ch}} \times \overline{h_{ch}} \times F_{in} \times F_{out} + (\overline{Size_{in}} + 3 \times \overline{Size_{acc}}) \times (F_{in} - 1) \times F_{out} \\ + (\overline{Size_{in}} + 2 \times \overline{Size_{out}} + \overline{Size_{acc}}) \times F_{out} \tag{5.10}$$

For layer groups without convolution, the size is:

$$Size_{sw\_op} = (\overline{Size_{in}} + 3 \times \overline{Size_{acc}}) \times (F_{in} - 1) \times F_{out} + (\overline{Size_{in}} + 2 \times \overline{Size_{out}}) \times F_{out} \tag{5.11}$$

All symbols with bars are typology parameters after channels folding. To convert $Size_{sw\_op}$ into numbers of bytes, results should be multiplied by 2, since 16-bit data type is used. Based on tests on the test platform, the latency of cache operations can be estimated by:

$$Latency_{cache} = \frac{Size_{sw\_op}}{10^6} \times 2 \times 0.9(ms) \tag{5.12}$$

"0.9" is the approximated slope in Figure 5.6.

Another part of software running time comes from setting typology parameters for IPs. In one layer group, IPs will be started for $F_{in} \times F_{out}$ times. By measuring on the board, the process takes 9 μs on average. Therefore,

$$Latency_{start} = 9 \times F_{in} \times F_{out} \ (\mu s) \tag{5.13}$$

$$Latency_{network\_sw} = Latency_{cache} + Latency_{start} \tag{5.14}$$

Finally, the overall latency estimation on software and hardware is:

$$Latency_{network} = Latency_{network\_hw} + Latency_{network\_sw} \tag{5.15}$$

Figure 5.7 demonstrates model predictions on some design points. The software latency model is helpful to describe real situations.

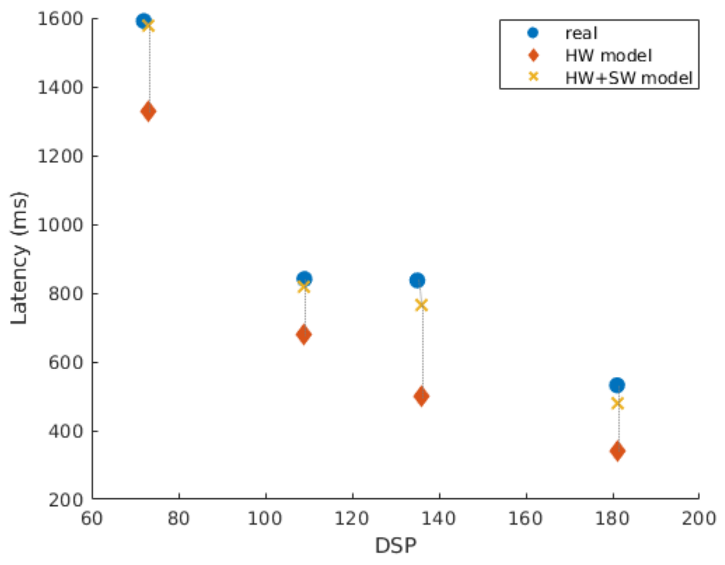


Figure 5.7: Model evaluation on certain design points

5.5.4 Design Space Exploration

Previous models give predictions on latency, BRAM and DSP utilisation. Every design parameter vector is mapped to a certain number of resources and latency. Parameters actually connect resources to performance, which forms the design space. By going over the design space, the configurability of the architecture is demonstrated. DSE offers an efficient way to find optimal points, which reach lowest latency with as few resources as possible.

Figure 5.8 and 5.9 illustrate design space exploration results of this design. Optimal points appear on the envelope of the graphs. For a given platform, maximal resources available are certain numbers. After applying the resource limitation, it is easy to determine the optimal design for a specific device.

Another finding from the graphs is that optimal points show an inverse proportional relationship. The slope of the fitting function decreases as resources increase. Therefore, doubling resources cannot bring the same amount of speed-up. There are other factors contributing to the latency like memory bandwidth and software costs. They are not affected when the computation capability of hardware IPs improves.

Design points of BRAM show a sparser distribution than DSP. In other words, it is easier to have bad decisions on BRAM and make them underutilised. In order to achieve high parallelism, arrays in HLS are partitioned for more memory ports. When the array is not big enough, each 18Kb BRAM fails to be fully filled. As a result, much memory space is wasted.

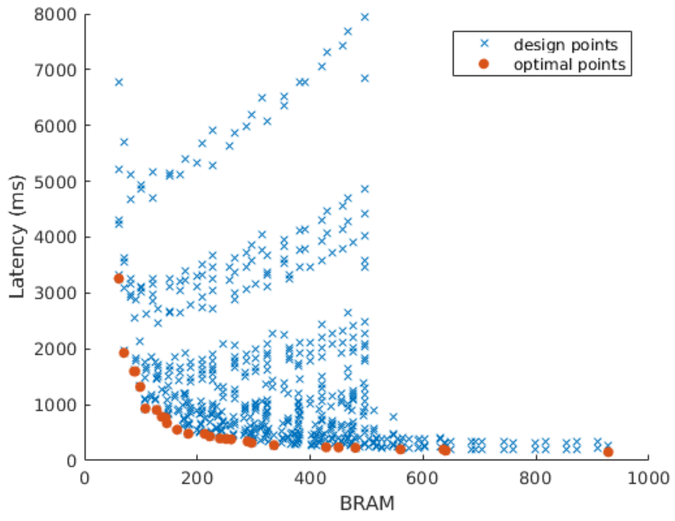


Figure 5.8: Design space exploration between BRAM and latency

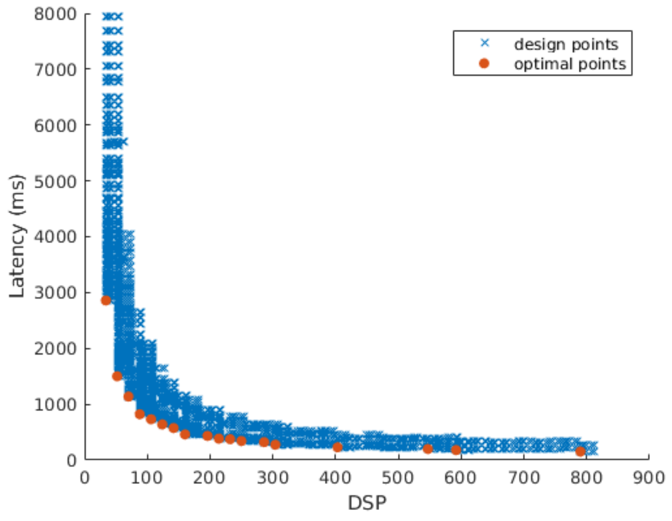


Figure 5.9: Design space exploration between DSP and latency

5.5.5 Model Deviation Analysis

To get an idea of models' accuracy, 30 design points are tested on the board. The prediction error of latency is within 9.8%, and deviation in resources is smaller than 5%. Test coverage is low because of long execution time of re-packaging IP, synthesis, place and route and generating bitstreams. For each test point, the target clock period in HLS should be adjusted until time constraints are met after place and route.

To summarise model analysis, it is necessary to explain reasons which may affect model accuracy. For the latency only, it is assumed before that latency is equal to pipeline initial interval multiplies tripcount. The condition is the product should be far larger than the pipeline depth. However, when the channels folding factor increases, hardware IPs will be launched for more times. It also means entering and exiting pipelines are happening more frequently. In this situation, the assumption is not solid.

Timing constraints also cause deviation in latency prediction. When the percentage of resource utilisation rises, route congestion will result in longer net delay [47]. It is then harder to schedule many operations within one clock cycle.

In order to meet setup or hold time constraints, the pipeline will have more stages. The depth of pipeline therefore increases. In addition, distance between dependent operations may become larger. Some operations have to wait for more cycles until inputs are available, which leads to a larger II.

Software latency model is based on results shown in Figure 5.6. Time of cache flush and invalidation is almost linear to memory size. But it is not a precise assumption especially when the memory size is smaller than 1 MB.

The model shows relatively accurate predictions on BRAM and DSP utilisation. The only problem is that the tool may choose to use a simpler implementation. LUTRAM rather than BRAM may be used when memory size is tiny. DSP can be replaced with LUT and FF, especially on the control path. The design can specify actual resources with HLS directives, but it also leads to some unreasonable resources utilisation.

Chapter 6

Evaluation

AFTER design space exploration, an optimal design point can be found for the specific test device. First, the performance of this design is compared with similar FPGA implementations of YOLO networks. Afterwards, the results are evaluated against CPU and GPU, in order to demonstrate the advantage of FPGA.

6.1 Comparison with existing FPGA implementations

In order to evaluate this work, five aspects are mainly considered, including target networks, platforms, resource utilisation, speed, and power consumption.

6.1.1 Target Network

For this design, the target network is YOLOv3-tiny. It includes 13 convolutional layers, 6 max pooling layers, 1 upsample layer, 2 route layers and 2 yolo layers. The workload of convolution only is 5.56 GOPs. Data and weights are quantised to 16 bits, while all intermediate values in convolution are kept as 32 bits. The network is trained for COCO-dev 2014 dataset. Tested on COCO-val5k, the fixed point version achieved 30.9% mAP50, compared with 33.4% of the original floating point network.

As no published result on FPGA implementation of YOLOv3 is discovered, all reference works are based on previous versions of YOLO. It is important to firstly mention network differences and how they will affect following evaluation.

In YOLO family, workload is used to describe operations of convolution only. However, the computation of other layer types should not be neglected. Therefore, workload here cannot reflect real amount of computation involved.

Another difference exists in the dataset. In the first and second generation of YOLO, networks are trained for VOC2007 [48] and VOC2012 [49] dataset. In YOLOv3 and also in this design, dataset used is COCO2014 [50]. VOC dataset only includes 20 classes but COCO dataset instead has 80 classes. As a result, YOLOV3 is more sensitive to data quantisation noise and needs more bits to maintain detection accuracy, shown in Table 6.1.

Table 6.1: Target networks comparison of the proposed design with previous works

	ref1 [34]	ref2 [51]	ref3 [52]	ref4 [38]	this work
Network	FPGA YOLO	Tincy YOLO	Lightweight YOLOv2	YOLOv2 tiny	YOLOv3 tiny
Image size	640×325	416×416	224×224	416×416	416×416
Data type	-	1-8b	1-32b	1-6b	16b
Workload (GOPs)	-	4.44	14.97	6.97	5.56

6.1.2 Platform

Test device used here is Zedboard development kit with Xilinx XC7Z020-CLG484-1 SoC and 512 MB DDR3 [53]. For the FPGA chip, the clock frequency of programmable logic and processing system is 100 MHz and 666.7MHz respectively. Inside PL, there are 280 BRAM, 220 DSP, 106k FF and 53k LUT available.

Clock frequency of FPGA affects performance. Higher clock frequency of PL means higher throughput, although the pipeline will be deeper to meet timing constraints. In terms of ARM processors, higher frequency leads to smaller instruction cycle. Software drivers can thereby run faster.

Although design space exploration is used and this design is not intended to optimise for a specific platform, many design decisions have already been made to ensure the

Table 6.2: Platforms comparison of the proposed design with previous works

	ref1	ref2	ref3	ref4	this work
Platform	-	XCZU3EG	ZCU102	Virtex-7 VC707	Zedboard
PL frequency (MHz)	-	-	300	200	100
PS frequency (MHz)	-	1500	1500	-	666.7
BRAM18k	787	-	1706	1026	185
DSP	409	-	377	168	160
LUT	47k	-	135k	86k	25.9k
FF	40k	-	370k	60k	46.7k

architecture can be tested on Zedborad. Such decisions limit performance and potential of the architecture.

For example, in this design, weights of each layer have been reloaded repeatedly rather than stored on FPGAs before run-time. Because BRAMs on Zedboard cannot accommodate all weights. A dynamic configurable architecture is adopted which takes layer group as the basic unit. However, in designs like ref3, the entire network is deployed on hardware. It requires a lot resources, but also reduces transactions between hardware and off-chip memory.

6.1.3 Speed & Resource Efficiency

For an embedded object detection, latency is the major concern to achieve real time. In this design only, latency and throughput mean the same thing, because each frame is executed in completely sequential order. However, if frames are pipelined, high throughput is not necessarily equivalent to low latency.

Table 6.3: Speed & resource efficiency comparison of the proposed design with previous works

	ref1	ref2	ref3	ref4	this work
Latency (ms)	52	63	25	9.15	532
Throughput (GOPS)	-	71.04	610.93	464.7	10.45
Efficiency (GOPS/DSP)	-	-	1.62	2.77	0.065
Efficiency (GOPS/BRAM)	-	-	0.36	0.45	0.056
Efficiency (GOPS/kLUT)	-	-	4.53	5.4	0.40
Efficiency (GOPS/kFF)	-	-	1.65	7.75	0.22

The problem is that resource efficiency ignores the scalability of resources. All reference works are not tunable between resources and performance. In other words, they are designed and optimised for a specific device, which is different from the aim of this work. To make design scalable, complex control logic with more multiplexes will be generated. Scalability also influences design decisions. For instance, explained in Chapter 4, one advantage of channels interleaving is making buffer size tunable. But uch decisions may not lead to optimal performance.

In addition, resource efficiency usually is not constant, as throughput rarely increases linearly with resources. Mathematically, if the relationship between resources and throughput is expressed as a function. Resource efficiency is only the derivative at an operating point. It is hard to determine which architecture is better by comparing derivatives at different points.

It is also important to consider the range of scalability. For instance, one design may achieve high resource efficiency over the whole design space. But it can only be tuned within a small range. These factors explain some reasons why this work does not seem very competitive with other works.

6.1.4 Power

Power mentioned here includes both the on-chip and off-chip components. For embedded applications, low power consumption is always desirable. Energy per frame and power efficiency are two metrics that combine speed and power. It is possible that a design might have a high power, but it is also able to run very fast. The device might sleep when it is idle to save energy. Under such circumstance, evaluating energy per frame or power efficiency is more meaningful.

The power consumption of Zedboard is calculated by the product of voltage supply and current drawn by the board. To get the current, the method is measuring the voltage across the current sense resistor. On Zedboard, the resistor is located at J21, which is 10 mΩ.

Table 6.4: Power comparison of the proposed design with previous works

	ref1	ref2	ref3	ref4	this work
Power (W)	7.518	6	4.5	18.29	3.36
Power efficiency (GOPS/W)	-	11.84	135.76	25.41	3.11
Energy per frame (J)	0.390	0.375	0.113	0.167	1.79

6.2 Comparison with CPU and GPU

One test runs on the laptop, which is equipped with Intel(R) Core(TM) i7-7500U CPU at 2.70GHz and 8GB DDR4 at 2133MHz. Within the test, only one core or more precisely one hyper-thread is used. The operating system is Ubuntu 16.04.6 LTS. This reference platform is called "Intel CPU" for the rest of this chapter.

The second test is only using the processing system of Zedboard, which is ARM-Cortex A9 running on bare-metal at 667MHz and the off-chip memory is 512MB DDR3. This test platform is called as "ARM CPU".

The programme used for testing is based on the original software implementation of YOLO. All file operations are replaced by storing weights and input images in header files. Same as the FPGA design, only the part of network prediction is evaluated, and others like getting bounding boxes and non maximum suppression (NMS) are removed.

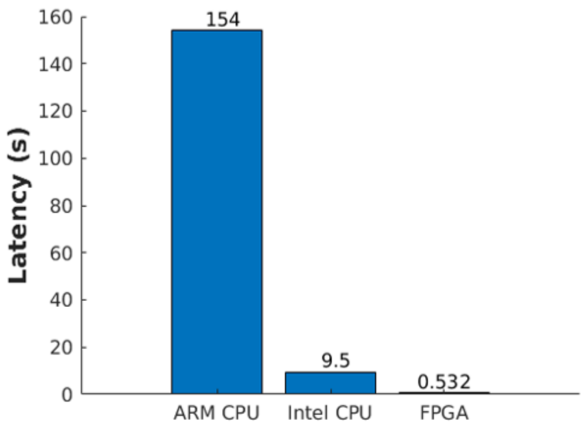


Figure 6.1: Latency comparison of CPU and FPGA

From Figure 6.1, this design on FPGA provides 289 and 17.9 times hardware acceleration compared with ARM and Intel CPU. The huge performance gap between two

CPUs also shows PS of Zedboard behaves very bad for this task. It may limit further acceleration of programmable logic.

Joseph Redmon et al. [12] deployed floating point YOLOv3-tiny on a Pascal Titan X which achieves 220 FPS. The minimal system power is 600 W [54]. Therefore, the power of efficiency is 2.03 GOPS/W. By comparison, this design is 1.53 times the GPU implementation in terms of power efficiency.

6.3 Summary

To summarise, this design provides obvious hardware acceleration for YOLOv3-tiny. It also has better power efficiency over GPU. Currently, this design is still not comparable to previous works on YOLO. The main reason is getting a tunable and dynamic configurable design sacrifices performance. Not only more multiplexes and complex control path are needed. But also design decisions are made to make the architecture scalable, which might not lead to optimal performance. Because of different datasets, it is also harder to quantise data for YOLOv3, which has a negative effect both on latency and resources.

Chapter 7

Conclusion

IN this thesis, a FPGA implementation of YOLOv3-tiny is given. It is based on the dynamic configurable architecture, and able to accelerate different layer groups without FPGA reconfiguration.

The convolutional layer uses multiply-accumulate batches to achieve parallelism among input and output channels. Batch normalisation is merged into the convolution by transforming weights before run-time. Channels are interleaved during DMA transfer to reduce the buffer size. Analysis of data and weight distribution is carried out to apply fixed point quantisation.

Accumulation & Activation module is responsible for accumulation when input channels folding happens. It also implements biasing and leaky ReLU activation. Both max pooling and upsample layer use a line buffer to support sliding window. The difference is that pooling compresses data while upsampling expands data. Inside YOLO layer, Xilinx fixed point exponential library is used for sigmoid activation. All hardware IPs are controlled by design parameters and typology parameters. So their latency and resources can be estimated.

In system design, the original YOLOv3-tiny network is transformed via layers combination, channels folding, channels padding and size padding. Afterwards, the network is forwarding as the unit of layer group. The corresponding hardware accelerator is a three-stage pipeline, connected to PS through two 64-bit DMA data ports and one AXI4-Lite control port.

Gathering analysis of all layers, a system model is finally proposed. It provides estimation on overall BRAM, DSP utilisation and latency. To get an accurate latency estimation, software costs on cache are also taken into account. Parameters bridge the gap between resources and latency. By traversing parameter choices, design space is explored. DSE establishes the trade-off between resources and latency. It also offers a concise way to find out optimal design points.

Finally, a design example on Zedboard is given to evaluate this architecture. Under the resource limitations, the achieved frame rate is 1.88 FPS and the throughput is 10.45 GOPS. Tested on COCOval5k dataset, mAP50 is 30.9% on this fixed point implementation without retraining weights. This design provides evident acceleration compared with CPUs. It also achieves higher power efficiency over GPU. Therefore, FPGA is promising to balance power consumption and low latency in embedded applications. At present, this design is not so competitive with previous works because of different typologies, test platforms, dynamic configurability and scalability.

In terms of future work, each layer type can provide several versions of implementation taking different strategies, as long as they have uniform interfaces. For example, one has channels interleaved while another is not. As a result, a larger design space can be exploited. It is reasonable because one decision is not always beneficial over the design space. Better results can be achieved for each design point without suffering too much penalty from a unified architecture.

For data quantisation, more thorough investigation should be carried out to further reduce data bitwidth. Logarithmic compression or binarised transformation are attractive, especially on layers responsible for extracting features. For layers predicting bounding boxes, keeping high precision is still preferable.

Bibliography

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 580–587, Jun. 2014.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 779–788.

[3] S. T. Chakradhar, M. Sankaradass, V. Jakkula, and H. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM Sigarch Computer Architecture News*, pp. 247–257, Jun. 2010.

[4] J. Walsh, N. O' Mahony, S. Campbell, A. Carvalho, L. Krpalkova, G. Velasco-Hernandez, S. Harapanahalli, and D. Riordan, "Deep learning vs. traditional computer vision," in *Computer Vision Conference (CVC) 2019*, Apr. 2019.

[5] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," *ACM Int. Symp. on Low Power Electronics and Design (ISLPED)*, pp. 326–331, 2016.

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on International Conference on Machine Learning*, Feb. 2015.

[7] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

[8] A. L. Maas, A. Y. Hannun, and A. Y. N, "Rectifier nonlinearities improve neural network acoustic models," *Proc. ICML*, vol. 30, 2013.

[9] R. Girshick, "Fast r-cnn," in *The IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015, pp. 1440–1448.

[10] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, Jun. 2015.

[11] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 6517–6525.

[12] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," Apr. 2018. [Online]. Available: https://pjreddie.com/media/files/papers/YOLOv3.pdf

[13] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. Reed, "Ssd: Single shot multibox detector," *European conference on computer vision*, pp. 21–37, Dec. 2015.

[14] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," *Advances in Neural Information Processing Systems*, May 2016.

[15] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 936–944.

[16] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal loss for dense object detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PP, pp. 1–1, Jul. 2018.

[17] H. Zhang, M. Xia, and G. Hu, "A multiwindow partial buffering scheme for fpga-based 2-d convolvers," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 2, pp. 200–204, Feb. 2007.

[18] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[19] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," *28th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 19–24, Jul. 2017.

[20] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 1800–1807.

[21] L. Bai, Y. Zhao, and X. Huang, "A cnn accelerator on fpga using depthwise separable convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, Oct. 2018.

[22] C. Zhu, S. Han, H. Mao, and W. Dally, "Trained ternary quantization," *ICLR*, 2017.

[23] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," *33rd International Conference on Machine Learning*, vol. 48, Nov. 2015.

[24] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2016, pp. 26–35.

[25] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *CoRR abs/1603.01025 (2016)*, Mar. 2016.

[26] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Computer Vision ECCV 2016: 14th European Conference*, Oct. 2016, pp. 525–542.

[27] S. I. Venieris and C. 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)*, May 2016, pp. 40–47.

[28] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design, ICCD 2013*, Oct. 2013, pp. 13–19.

[29] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*, Feb. 2015, pp. 161–170.

[30] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. sun Seo, and Y. 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*, Feb. 2016, pp. 16–25.

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

[32] S. I. Venieris and C. Bouganis, "Latency-driven design for fpga-based convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8.

[33] A. Rahman, S. Oh, J. Lee, and K. Choi, "Design space exploration of fpga accelerators for convolutional neural networks," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2017, pp. 1147–1152.

[34] G. Wei, Y. Hou, Q. Cui, G. Deng, X. Tao, and Y. Yao, "Yolo accelration using fpga architecture," in *2018 IEEE/CIC International Conference on Communications in China (ICCC)*, Aug. 2018, pp. 734–735.

[35] B. Liu and X. Xu, "Fclnn: A flexible framework for fast cnn prototyping on fpga with opencl and caffe," in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 238–241.

[36] Y. J. Wai, Z. bin, S. Irwan, and L. Kim, "Fixed point implementation of tiny-yolo-v2 using opencl on fpga," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, Jan. 2018.

[37] H. Nakahara and T. Sasao, "A high-speed low-power deep neural network on an fpga based on the nested rns: Applied to an object detector," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[38] H. K. Duy Thanh Nguyen, Tuan Nghia Nguyen, "A high-throughput and power-efficient fpga implementation of yolo cnn for object detection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.

[39] Xilinx, "Vivado design suite axi reference guide: High-level synthesis ug1037 (v4.0)," Jul. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[40] C. Zhu, S. Han, H. Mao, and W. Dally, "Optimizing cnn-based object detection algorithms on embedded fpga platforms," *International Symposium on Applied Reconfigurable Computing*, vol. ARC2017, pp. 255–267, 2017.

[41] A. Finnerty and H. Ratigner, "Reduce power and cost by converting from floating point to fixed point wp491 (v1.0)," Mar. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf

[42] Xilinx, "Vivado design suite user guide: High-level synthesis ug902 (v2019.1)," Jul. 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf

[43] Z. Song, Z. Liu, C. Wang, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, Sep. 2017.

[44] Xilinx, "7 series fpgas memory resource user guide ug473 (v1.14)," Jul. 2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[45] W.-T. Shiue, S. Udayanarayanan, and C. Chakrabarti, "Data memory design and exploration for low-power embedded systems," *ACM Trans. Design Autom. Electr. Syst.*, vol. 6, pp. 553–568, Oct. 2001.

[46] Xilinx, "Axi4-stream infrastructure ip suite v3.0 logicore ip product guide," Dec. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf

[47] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in fpga high-level synthesis," *CoRR*, vol. abs/1905.03852, Jun. 2019.

[48] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results," http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[49] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results," http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[50] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: http://arxiv.org/abs/1405.0312

[51] T. B. Preuer, G. Gambardella, N. Fraser, and M. Blott, "Inference of quantized neural networks on heterogeneous all-programmable devices," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 833–838.

[52] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga," in *2018 ACM/SIGDA International Symposium*, Feb. 2018, pp. 31–40.

[53] AVNET, "Zedboard (zynq evaluation and development) hardware users guide," Jan. 2014. [Online]. Available: http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[54] Nvidia, "Geforce gtx titan x user guide," 2014. [Online]. Available: https://www.nvidia.com/content/geforce-gtx/GTX_TITAN_X_User_Guide.pdf

Appendix A

YOLOv3-tiny Details

A.1 Network Structure

Table A.1: Yolov3-tiny structure details

layer	kernel (num)	kernel (size)	stride	inputs	outputs	workload (BFLOPs)
conv	16	3×3	1	$416 \times 416 \times 3$	$416 \times 416 \times 16$	0.150
max		2×2	2	$416 \times 416 \times 16$	$208 \times 208 \times 16$	
conv	32	3×3	1	$208 \times 208 \times 16$	$208 \times 208 \times 32$	0.399
max		2×2	2	$208 \times 208 \times 32$	$104 \times 104 \times 32$	
conv	64	3×3	1	$104 \times 104 \times 32$	$104 \times 104 \times 64$	0.399
max		2×2	2	$104 \times 104 \times 64$	$52 \times 52 \times 64$	
conv	128	3×3	1	$52 \times 52 \times 64$	$52 \times 52 \times 128$	0.399
max		2×2	2	$52 \times 52 \times 128$	$26 \times 26 \times 128$	
conv	256	3×3	1	$26 \times 26 \times 128$	$26 \times 26 \times 256$	0.399
max		2×2	2	$26 \times 26 \times 256$	$13 \times 13 \times 256$	
conv	512	3×3	1	$13 \times 13 \times 256$	$13 \times 13 \times 512$	0.399
max		2×2	1	$13 \times 13 \times 512$	$13 \times 13 \times 512$	
conv	1024	3×3	1	$13 \times 13 \times 512$	$13 \times 13 \times 1024$	1.595
conv	256	1×1	1	$13 \times 13 \times 1024$	$13 \times 13 \times 256$	0.089
conv	512	3×3	1	$13 \times 13 \times 256$	$13 \times 13 \times 512$	0.399
conv	255	1×1	1	$13 \times 13 \times 512$	$13 \times 13 \times 255$	0.044
yolo						
route				(13)		
conv	128	1×1	1	$13 \times 13 \times 256$	$13 \times 13 \times 128$	0.011
upsample			$2\times$			
route				(19,8)		
conv	256	3×3	1	$26 \times 26 \times 384$	$26 \times 26 \times 256$	1.196
conv	255	1×1	1	$26 \times 26 \times 256$	$26 \times 26 \times 255$	0.088
yolo						

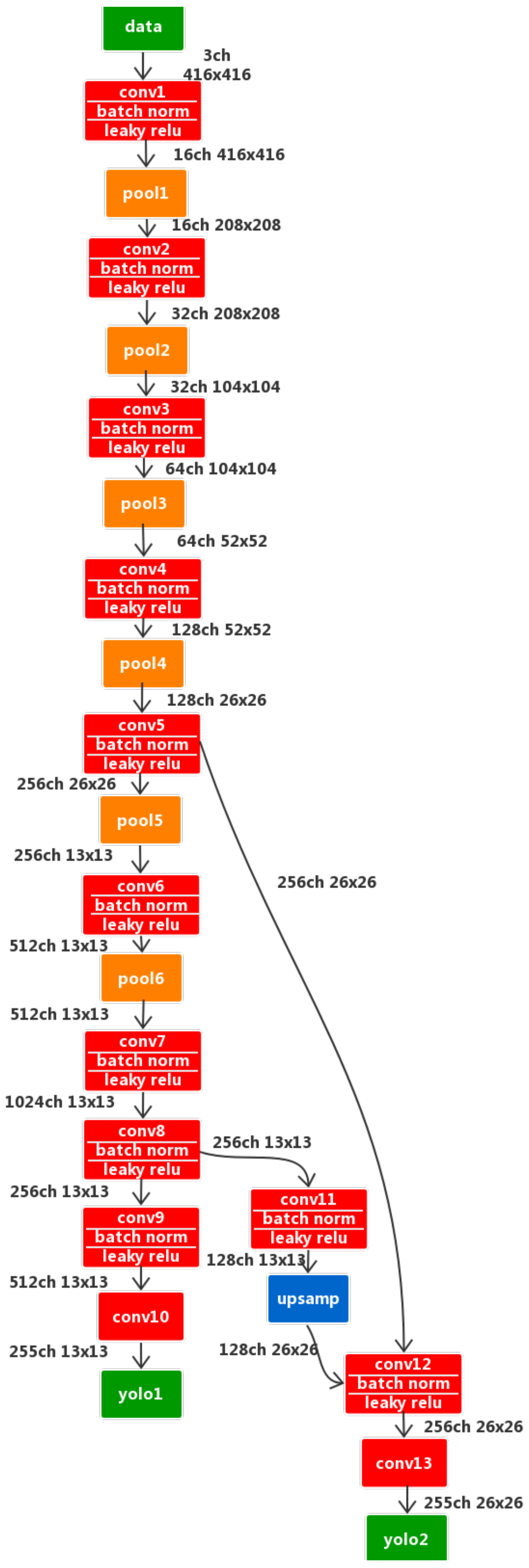
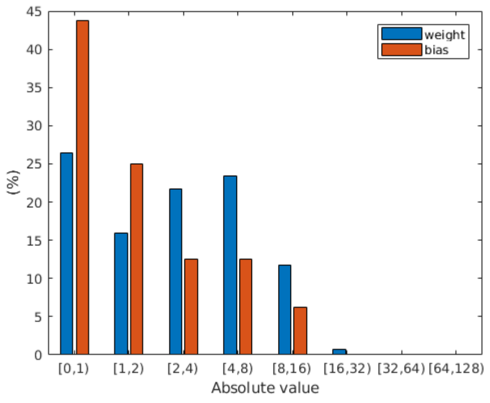


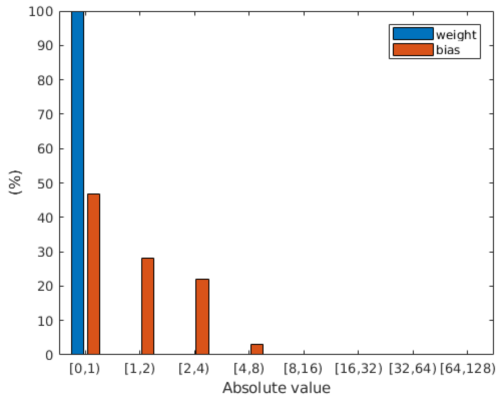
Figure A.1: YOLOv3-tiny typology

A.2 Weight Distribution

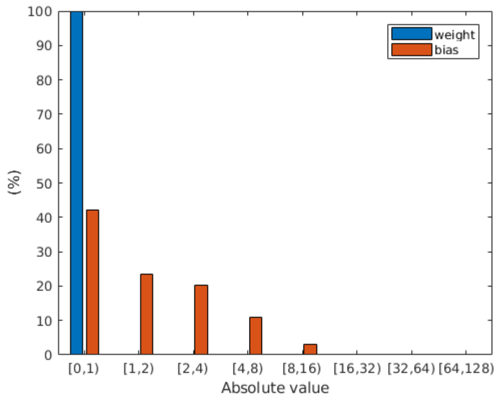
Convolution weights and biases distribution analysis after merging batch normalisation.



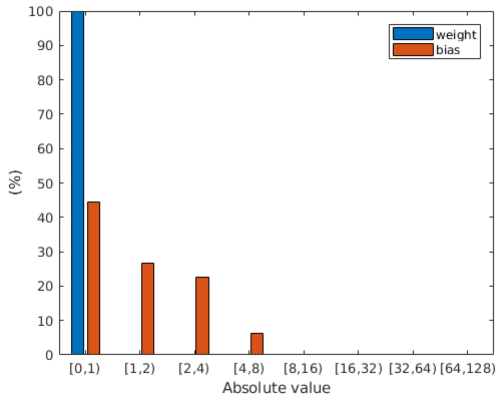
(a) conv1



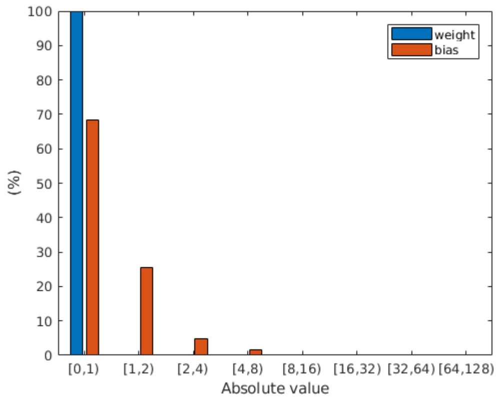
(b) conv2



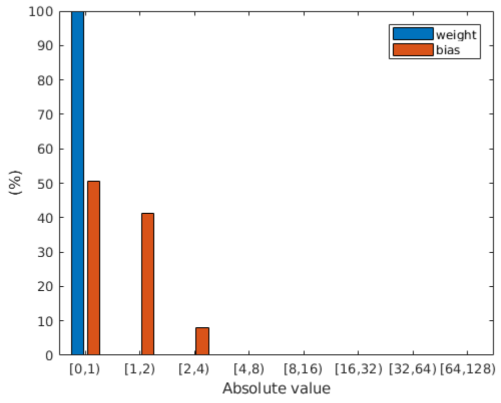
(c) conv3



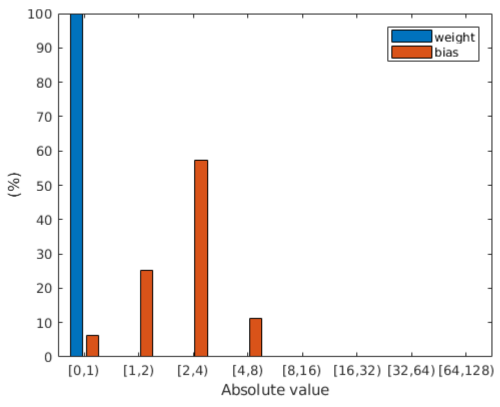
(d) conv4



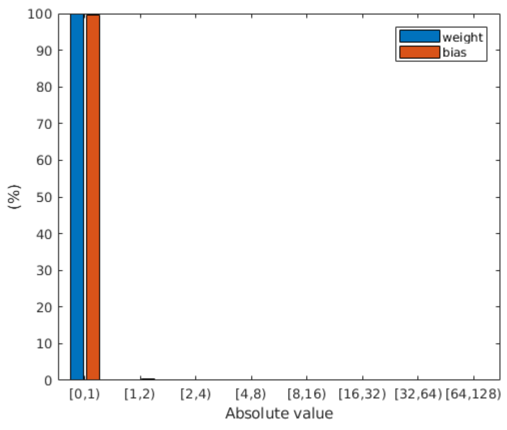
(e) conv5



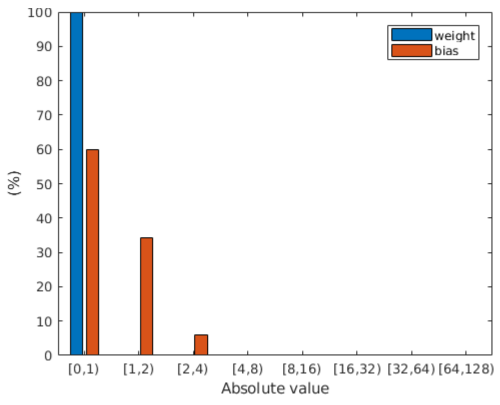
(f) conv6



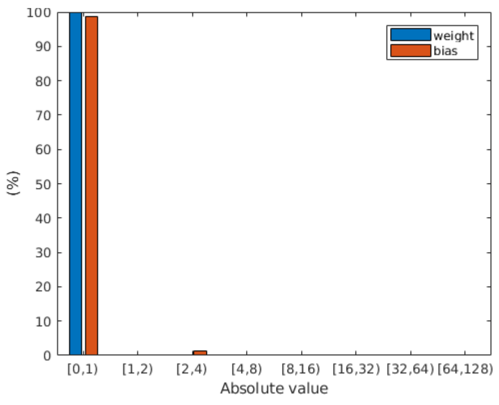
(g) conv7



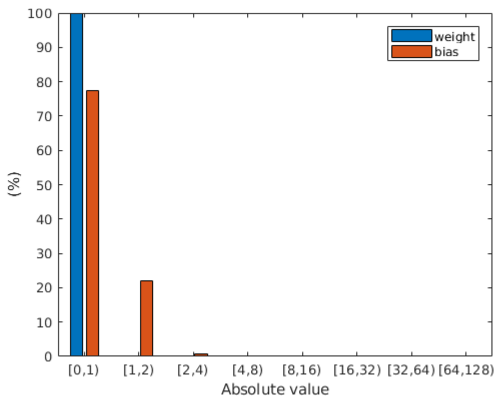
(h) conv8



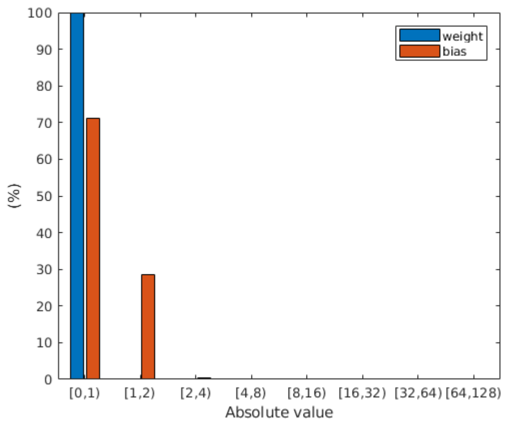
(i) conv9



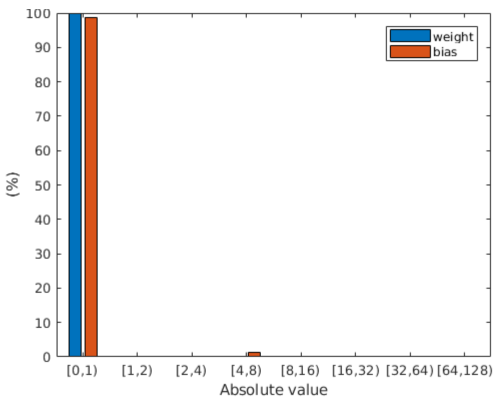
(j) conv10



(k) conv11



(l) conv12

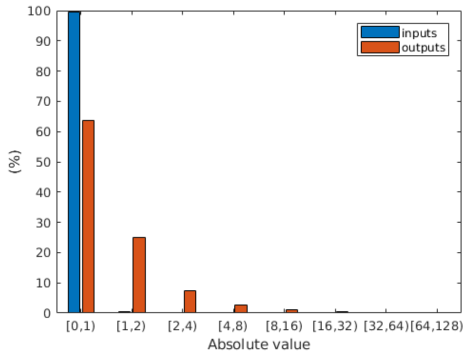


(m) conv13

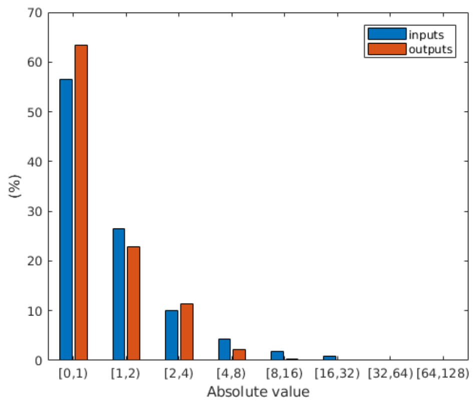
Figure A.2: Weights and biases distribution after merging BN

A.3 Data Distribution

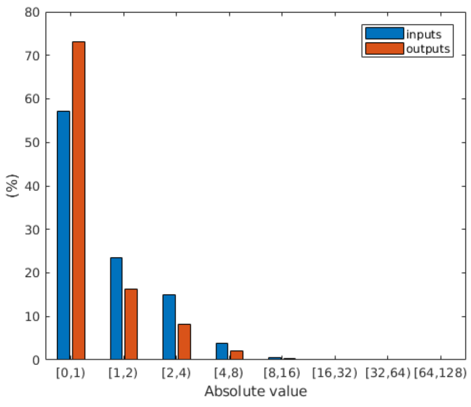
Inputs and outputs of all convolutional layers. Tests are carried on COCOval5k.



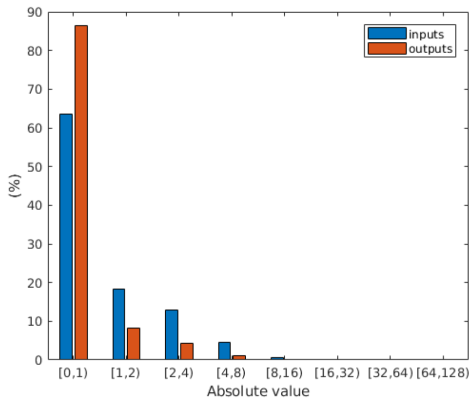
(a) conv1



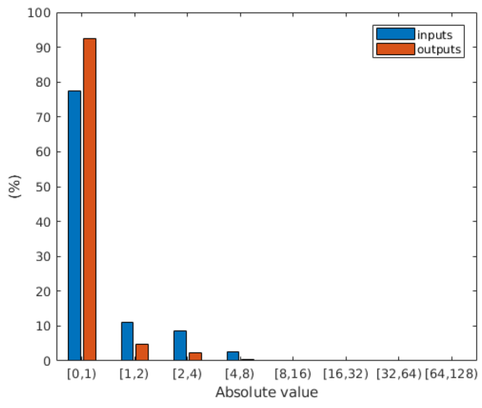
(b) conv2



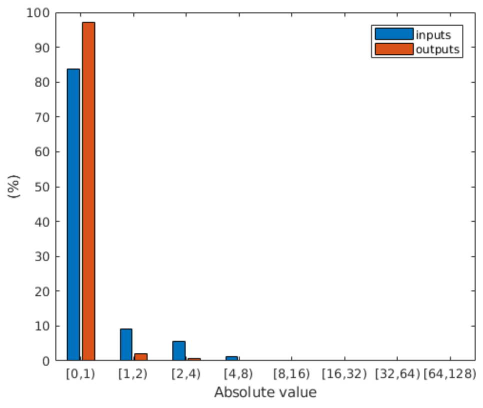
(c) conv3



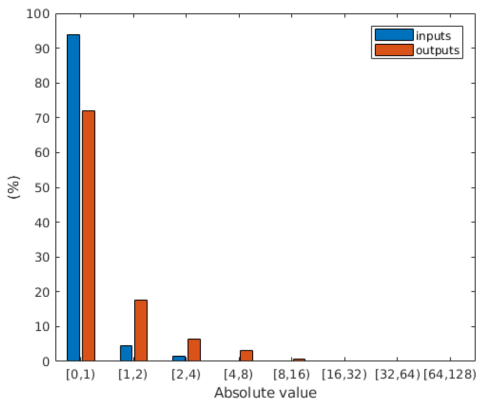
(d) conv4



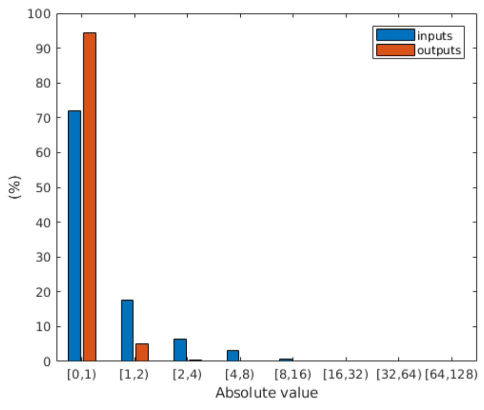
(e) conv5



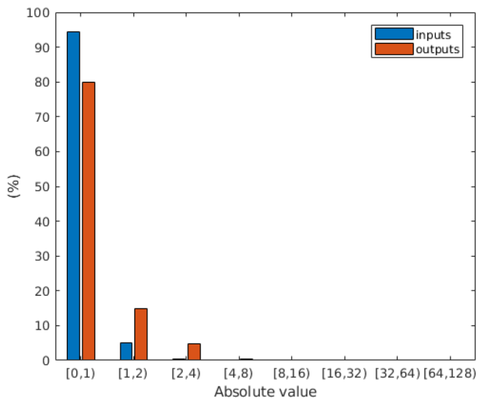
(f) conv6



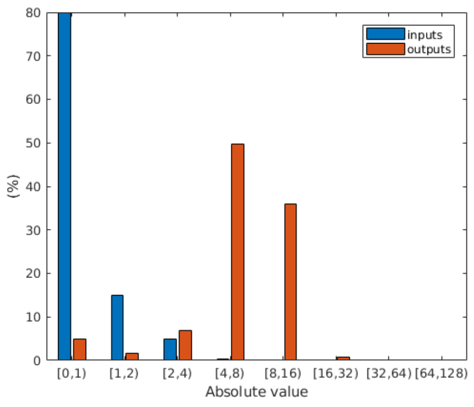
(g) conv7



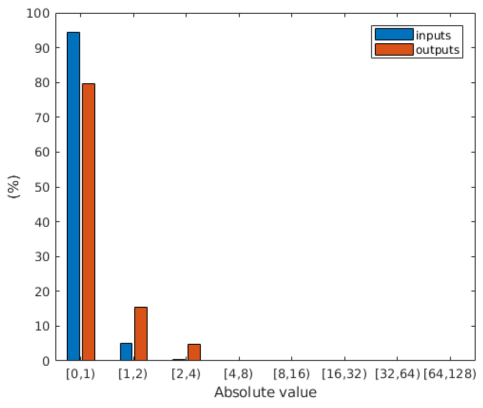
(h) conv8



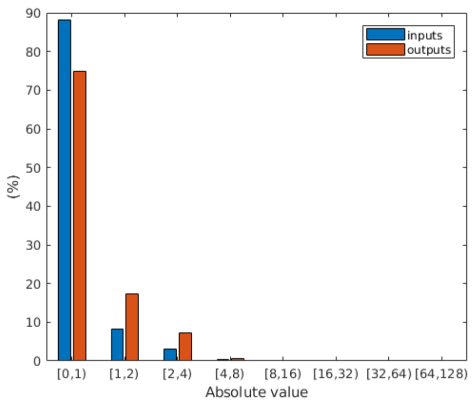
(i) conv9



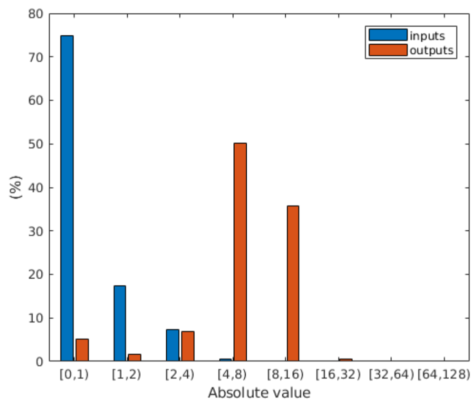
(j) conv10



(k) conv11



(l) conv12



(m) conv13

Figure A.3: Data distribution based on COCO-val5k

A.4 Floating Point predictions on COCOval5k

Table A.4: Floating point predictions on COCOval5k

metric	IoU	area	maxDets	result
Average Precision (AP)	0.50:0.95	all	100	0.154
Average Precision (AP)	0.50	all	100	0.334
Average Precision (AP)	0.75	all	100	0.124
Average Precision (AP)	0.50:0.95	small	100	0.045
Average Precision (AP)	0.50:0.95	medium	100	0.153
Average Precision (AP)	0.50:0.95	large	100	0.253
Average Recall (AR)	0.50:0.95	all	1	0.171
Average Recall (AR)	0.50:0.95	all	10	0.275
Average Recall (AR)	0.50:0.95	all	100	0.301
Average Recall (AR)	0.50:0.95	small	100	0.094
Average Recall (AR)	0.50:0.95	medium	100	0.340
Average Recall (AR)	0.50:0.95	large	100	0.480

A.5 Fixed Point predictions on COCOval5k

Table A.5: Fixed point predictions on COCOval5k

metric	IoU	area	maxDets	result
Average Precision (AP)	0.50:0.95	all	100	0.147
Average Precision (AP)	0.50	all	100	0.309
Average Precision (AP)	0.75	all	100	0.119
Average Precision (AP)	0.50:0.95	small	100	0.017
Average Precision (AP)	0.50:0.95	medium	100	0.127
Average Precision (AP)	0.50:0.95	large	100	0.281
Average Recall (AR)	0.50:0.95	all	1	0.162
Average Recall (AR)	0.50:0.95	all	10	0.225
Average Recall (AR)	0.50:0.95	all	100	0.229
Average Recall (AR)	0.50:0.95	small	100	0.022
Average Recall (AR)	0.50:0.95	medium	100	0.186
Average Recall (AR)	0.50:0.95	large	100	0.453

Appendix B

System Results

B.1 Vivado Block Design

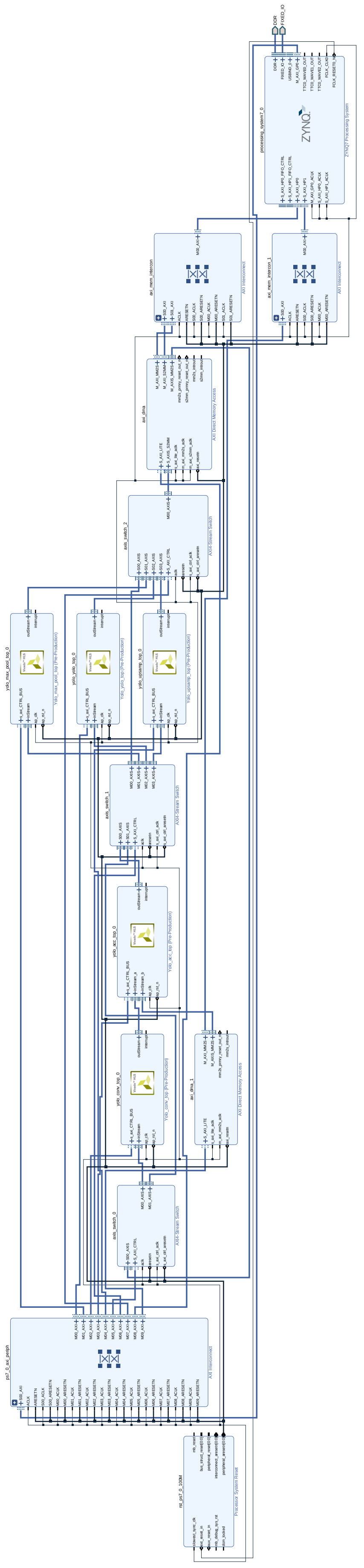
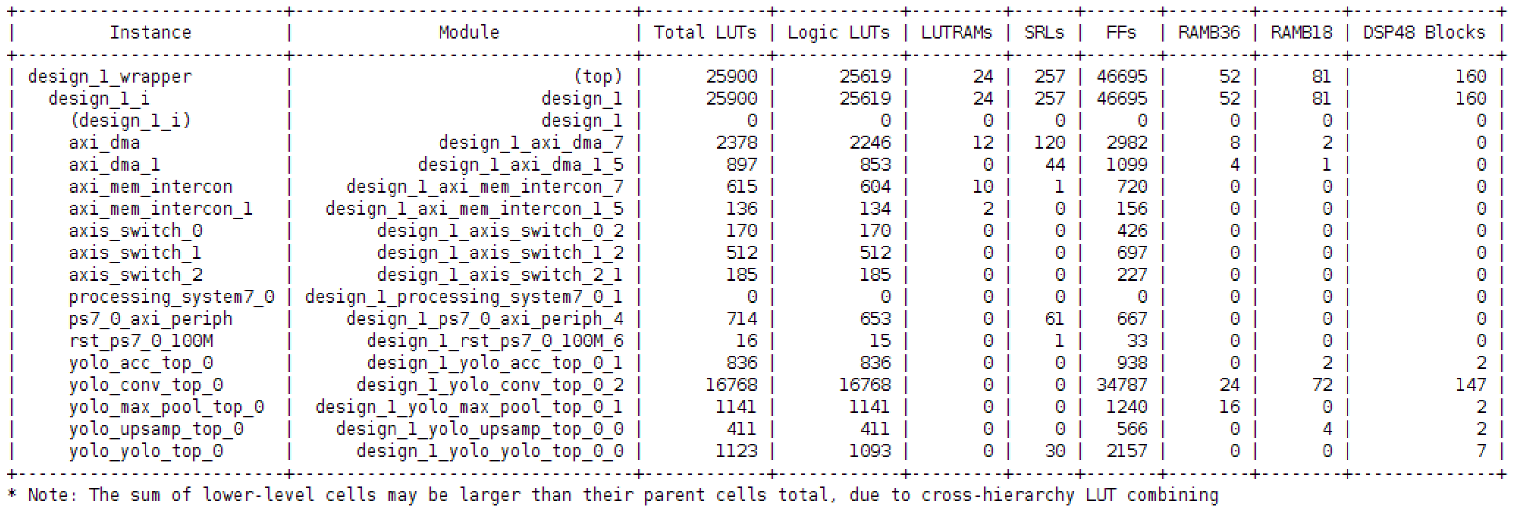


Figure B.1: Vivado block design

B.2 Vivado Resource Report



```
+--------------------------+-----------------------------------+------------+-------------+---------+------+-------+--------+--------+--------------+
|         Instance         |              Module               | Total LUTs | Logic LUTs  | LUTRAMs | SRLs |  FFs  | RAMB36 | RAMB18 | DSP48 Blocks |
+--------------------------+-----------------------------------+------------+-------------+---------+------+-------+--------+--------+--------------+
| design_1_wrapper         |                             (top) |      25900 |       25619 |      24 |  257 | 46695 |     52 |     81 |          160 |
|   design_1_i             |                          design_1 |      25900 |       25619 |      24 |  257 | 46695 |     52 |     81 |          160 |
|     (design_1_i)         |                          design_1 |          0 |           0 |       0 |    0 |     0 |      0 |      0 |            0 |
|     axi_dma              |                  design_1_axi_dma_7 |       2378 |        2246 |      12 |  120 |  2982 |      8 |      2 |            0 |
|     axi_dma_1            |                design_1_axi_dma_1_5 |        897 |         853 |       0 |   44 |  1099 |      4 |      1 |            0 |
|     axi_mem_intercon     |          design_1_axi_mem_intercon_7 |        615 |         604 |      10 |    1 |   720 |      0 |      0 |            0 |
|     axi_mem_intercon_1   |        design_1_axi_mem_intercon_1_5 |        136 |         134 |       2 |    0 |   156 |      0 |      0 |            0 |
|     axis_switch_0        |             design_1_axis_switch_0_2 |        170 |         170 |       0 |    0 |   426 |      0 |      0 |            0 |
|     axis_switch_1        |             design_1_axis_switch_1_2 |        512 |         512 |       0 |    0 |   697 |      0 |      0 |            0 |
|     axis_switch_2        |             design_1_axis_switch_2_1 |        185 |         185 |       0 |    0 |   227 |      0 |      0 |            0 |
|     processing_system7_0 | design_1_processing_system7_0_1 |          0 |           0 |       0 |    0 |     0 |      0 |      0 |            0 |
|     ps7_0_axi_periph     |          design_1_ps7_0_axi_periph_4 |        714 |         653 |       0 |   61 |   667 |      0 |      0 |            0 |
|     rst_ps7_0_100M       |            design_1_rst_ps7_0_100M_6 |         16 |          15 |       0 |    1 |    33 |      0 |      0 |            0 |
|     yolo_acc_top_0       |            design_1_yolo_acc_top_0_1 |        836 |         836 |       0 |    0 |   938 |      0 |      2 |            2 |
|     yolo_conv_top_0      |           design_1_yolo_conv_top_0_2 |      16768 |       16768 |       0 |    0 | 34787 |     24 |     72 |          147 |
|     yolo_max_pool_top_0  |       design_1_yolo_max_pool_top_0_1 |       1141 |        1141 |       0 |    0 |  1240 |     16 |      0 |            2 |
|     yolo_upsamp_top_0    |         design_1_yolo_upsamp_top_0_0 |        411 |         411 |       0 |    0 |   566 |      0 |      4 |            2 |
|     yolo_yolo_top_0      |           design_1_yolo_yolo_top_0_0 |       1123 |        1093 |       0 |   30 |  2157 |      0 |      0 |            7 |
+--------------------------+-----------------------------------+------------+-------------+---------+------+-------+--------+--------+--------------+
* Note: The sum of lower-level cells may be larger than their parent cells total, due to cross-hierarchy LUT combining
```

Figure B.2: Vivado resource hierarchy analysis