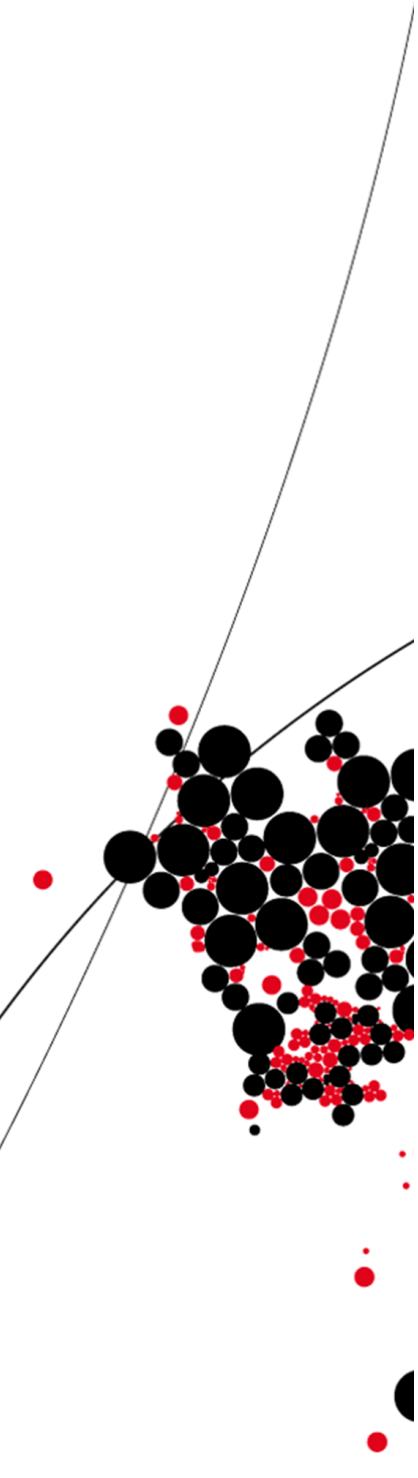
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

A Tunable Accelerator for the YOLOv4-tiny Object Detector using Vitis Unified Software Platform

Sridhar Balamurali

MSc. Thesis
May 2023



Supervisors
dr. ir. S.H. Gerez
dr. ir. N. Alachiotis
dr. C.G. Zeinstra

Computer Architectures and Embedded Systems Group

Faculty of Electrical Engineering,
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Deep learning-based object detection plays a crucial role in various computer vision applications. However, deploying these complex networks on embedded systems that have limited resources presents significant challenges. This thesis addresses these challenges by developing a unified and tunable FPGA accelerator, drawing inspiration from an existing architecture [1]. The accelerator is designed to optimize the performance of not only the computationally intensive convolutional layers but also other non-convolutional layers in the YOLOv4-tiny model. To achieve this, Vivado HLS is utilized to create IPs with design parameters that allow for tuning through parallelism control. Key optimizations, such as fixed-point quantization and channel interleaving, are employed. The Vitis unified software platform is utilized to dynamically configure the layers of YOLOv4-tiny within the processing system of the ZedBoard. The proposed accelerator achieves a significant speedup, with the convolutional layers running 20 times faster compared to previous works on the same platform with a MACC operation taking 2 clock cycles inside the convolution block with a throughput of 5.84 GOPS/secs, resulting in an inference rate of 3.3 seconds per image. The overall architecture achieves a throughput of 2.05 GOPS/sec with resource utilization of 166 (76%) DSP units, 149 (53%) BRAM18K blocks, LUT, and FF utilization of about 56% and 43% respectively. Furthermore, when compared with the ARM A9 processor on the ZedBoard and host CPU implementation, the implemented architecture demonstrates a speed improvement of 58x, and 3x respectively.

Contents

List of acronyms

DNN	Deep Neural Network
YOLO	You Only Look Once
GPUs	Graphics Processing units
FPGAs	Field Programmable Gate Arrays
IoU	Intersection over Union
FPS	Frame Per Seconds
PE	Processing Element
PS	Processing System
PL	Programmable Logic
PL	Programmable Logic
MACC	Multiply and Accumulate
mAP	Mean Average Precision
CNN	Convolutional Neural Network
COCO	Commom Objects in Context
HLS	High Level Synthesis
RTL	Register Transfer Level
ReLU	Rectified Linear Unit
AXI	Advanced eXtensible Interface
IP	Intellctual Property
TFLM	TensorFlow Lite Micro
SPP	Spatial Pyramid Pooling

Chapter 1

Introduction

In today's information age, image processing is a crucial technology that enables electronic systems to perceive, analyze, and manipulate the world. Traditional image processing techniques primarily rely on mathematical algorithms or feature descriptors [2]. However, since the early 2000s, there has been a growing interest in bio-inspired deep neural network (Deep Neural Network (DNN)) applications. These DNNs have become increasingly popular due to their ability to learn complex patterns and features from large datasets, leading to superior performance in various image-processing tasks.

In the present day, computer vision is a dynamic area of research that is producing impressive outcomes. An extensively employed task in computer vision is the detection of objects. This task facilitates systems to identify and categorize objects within images. Conventional object detection methods relied on manually crafted feature extractors, but they have been surpassed by deep learning techniques [3] [4]. One of these approaches that have achieved real-time performance in object detection is the You Only Look Once (YOLO) (You Only Look Once) detector, which was introduced in 2016 [5]. YOLO provides a novel method where image pixels are used to predict object locations and corresponding classes. Earlier techniques utilized complex pipelines that were difficult to optimize and performed relatively poorly. Numerous versions of YOLO have been released over the years, but this work employs the most recent scientifically validated version, which is YOLOv4-tiny [6].

The primary difficulty with DNNs like You Only Look Once (YOLOv4-tiny) is their prolonged computational time, which necessitates high-processing-capability platforms. Generally, greater computational ability results in increased power consumption. Graphics Processing units (GPUs) and large-scale cloud servers are typical examples of such platforms.

Researchers are exploring balancing performance and power consumption for deploying neural networks in embedded applications. Field Programmable Gate Arrays (FPGAs) have shown promise as a platform for this purpose, as they can be hardware reconfigured to enable flexible parallelism [7]. Additionally, many hardware acceleration techniques make FPGAs suitable for energy-efficient scenarios. These include utilizing specialized hardware

blocks, such as DSPs and BRAMs, to accelerate computation, as well as optimizing memory usage with techniques like data folding and line buffering. Overall, FPGAs offer a potential solution to the challenge of deploying DNNs in embedded applications with limited resources.

In recent times, there has been a surge of network architectures that are highly efficient in object detection, which have the potential to advance the development of intelligent systems. However, implementing these networks on FPGA poses a considerable challenge due to their large scale. Moreover, it is highly desirable to have a tunable FPGA implementation available for these complex networks. By providing a tunable design, different design decisions can be made to cater to the varying requirements of different applications without having to start the system design process from scratch repeatedly.

1.1 Problem statement and System Setup

The goal of this thesis is to develop a tunable YOLOv4-tiny FPGA implementation targetting resource-constrained devices like Zedboard, which is an ARM/FPGA Soc development board. There have been recent works that accelerate the YOLOv4-tiny object detection algorithm. A Convolutional Neural Network (CNN) accelerator for YOLOv4-tiny integrated with TensorFlow lite framework (TensorFlow Lite Micro (TFLM)) has been implemented using a Catapult high-level synthesis tool in order to realize real-time performance [8]. An accelerator was developed for FPGA to increase the throughput for the convolutional layers of YOLOv4-tiny to improve upon the DSP utilization of the design [9]. Both designs used Catapult C for High-level synthesis.

Catapult can be a useful tool for accelerating FPGA design but it has some limitations too [10]. Catapult does not provide the level of control and optimization that is available in traditional RTL design flow provided by Xilinx tools such as Vivado. This makes it less suitable for designs that require fine-grained optimization and control over the hardware implementation. This fine-grained optimization is required when making the design more tunable. Moreover, Catapult offers limited support for the AXI interface, which makes it difficult to interface with other Intellctual Property (IP) blocks or to implement complex data transfer between the processing system and the programmable logic blocks inside FPGA. Therefore for this project Xilinx tool such as Vivado HLS is used to implement IP blocks. These IP blocks are integrated using the Vivado designer suite. Finally, Vitis unified software platform is used for creating hardware software codesign implementation. Further details regarding how Vivado HLS can be utilized to create Advanced eXtensible Interface (AXI) interfaces and how other directives can be utilized to improve the performance of the accelerator is discussed in chapter 4.

A bare-metal application without a camera is chosen for this implementation because it eliminates the need for an extra operating system, which results in lower overhead and better system performance. Additionally, it provides greater control over hardware resources,

leading to more efficient utilization of available resources. Finally, the use of bare-metal implementation enables greater customization and flexibility in terms of hardware-software integration, allowing the system to be adapted to specific needs. Images are firstly pre-processed on the host PC. Pre-processing involves the following steps:

1. Image resizing
2. Letterboxing the image.
3. Channel Interleaving.

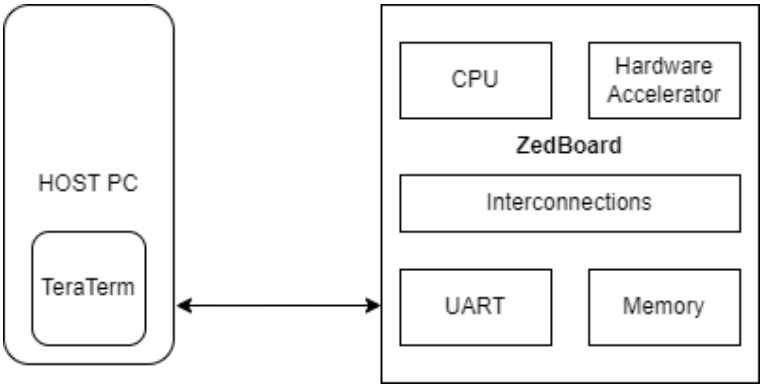


Figure 1.1: System Overview for YOLOv4-tiny Algorithm Processing [8]

These pre-processed images are fed to the accelerator architecture running on the FPGA programmable logic. The Zedboard outputs bounding box coordinates and class predictions along with their accuracy, which is then relayed back to the host PC through the UART interface and displayed on a serial terminal called "Tera Term".

1.2 Research Questions

How can a tuneable FPGA design be created for the deep-learning object detector YOLOv4-tiny with possible opportunity for carrying out design space exploration?

To answer this main research question, the following research subquestions need to be answered:

- How can a YOLOv4-tiny be optimized at the software level before hardware accelerating the model?
- What are the design decisions that are taken to accelerate the YOLOv4-tiny model?
- How can an FPGA design for YOLOv4-tiny be made parameterisable? Given the parameters what values are optimal?
- How can a YOLOv4-tiny accelerator be created using the Vitis unified software platform?

- Can a design space exploration be carried out for the YOLOv4-tiny model to find optimal design points which reach low latency with as few resources as possible?

1.3 Contribution

The primary aim of this thesis, as stated in the central research question, is to develop a flexible FPGA design for YOLOv4-tiny using the Vitis Unified Software Platform on limited-resource devices. However, this is not the only contribution of this work. The main contribution of this thesis has been listed below.

- Bare-metal software applications, capable of running on resource-limited platforms like Zedboard, have been developed by integrating the Darknet framework with both floating and fixed-point operations. These applications serve as a foundation for executing neural networks on a single-core processor
- An FPGA implementation of the YOLOv4-tiny neural network, implemented through Xilinx HLS tools is provided. The FPGA hardware IPs are utilized as a dedicated CNN accelerator. Fixed-point quantization and channel interleaving techniques have been employed to enhance acceleration performance, taking into account the network topology and data analysis.
- The presented work proposes a dynamic and configurable architecture, capable of computing all layers in the YOLOv4-tiny neural network without the need for bitstream reconfiguration. The technique utilized is known as network transformation, which involves reshaping the original network structure to be compatible with the accelerator. This approach facilitates efficient computation of the neural network on FPGA hardware.

The distinction between *design parameters* and *typology parameters* is important in the context of hardware design for neural networks. Design parameters are concerned with the hardware details such as the number of memory banks, the maximum number of channels the IP can process, and other related aspects. In contrast, typological parameters are related to the network typology and include information about the layers and their configurations. The rest of the thesis will make use of these two concepts to design and implement FPGA-based hardware for neural networks.

1.4 Thesis Outline

Chapter 2 focuses on providing background information about Convolutional Neural Networks and their typical components. The chapter also explains why YOLOv4-tiny was chosen as the network of interest for this particular design.

Chapter 3, popular techniques for optimizing Convolutional Neural Networks on Field Programmable Gate Arrays (FPGAs). It also highlights recent research studies that have ex-

plored methods for accelerating YOLO on FPGAs.

Chapter 4 delves into the software tools and platforms used to implement the various layers of the YOLOv4-tiny network. The chapter provides a detailed explanation of the Vitis Unified software platform and its role in implementing CNN. The chapter gives the reasons for choosing Vitis Unified, highlighting its advantages over other software platforms in terms of ease of use, flexibility, and performance. The chapter also presents the design flow of the Vitis platform and the optimization techniques used to improve the performance of the implemented layers. Overall, the chapter provides a comprehensive understanding of the software tools used in the implementation of the YOLOv4-tiny. network.

Chapter 5 provides a comprehensive description of the bare-metal floating-point model of the YOLOv4-tiny network, including its architecture and layers. It discusses the results obtained from profiling the simulation model to identify the bottleneck layers that may affect the network's overall performance. Additionally, the chapter introduces the bare-metal fixed-point model of the YOLOv4-tiny network, which serves as a reference for hardware implementation. The section also explains the methodology used to convert the floating-point model into a fixed-point model and describes the differences between the two models. Overall, Chapter 5 provides important background information necessary for the subsequent hardware implementation of the network.

Chapter 6 focuses on the hardware IP design of each layer in the YOLOv4-tiny network. It begins by providing an overview of the main methods and algorithms for each IP. The chapter then goes on to provide a detailed explanation of the design and implementation process.

Chapter 7 concentrates on the system-level design and optimizations for the implementation of the YOLOv4-tiny network on FPGA.

Chapter 8 evaluates the results of the hardware-software codesign model and compares them with previous work regarding YOLOv4-tiny. It also compares the FPGA implementation results of the model with those of the CPU.

Chapter 9 summarizes the key findings of the thesis and highlights the contributions of the work. It also discusses the limitations of the proposed approach and suggests potential areas for future research and improvement.

Chapter 2

CNN Background

Background information about neural networks is provided in this chapter to support the selection of YOLO. Section 2.1.1 explains the fundamental concepts of Convolution Neural Networks which include the metrics for evaluating CNN and datasets used. Section 2.2 introduces Yolo Algorithm. Section 2.3 describes the motivation for the selection of the YOLOv4-tiny object detection algorithm. Finally, section 2.4 gives the description of YOLOv4-tiny architecture.

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 organization. Because each neuron only responds to a certain receptive field. The reduction in connectivity means lower computational complexity. Therefore, CNN has gained popularity in image classification, object detection, and signal processing.

2.1.1 Convolutional Layer

Convolution filters are the main computational element in CNNs. In the case of an RGB image input, there are three input channels, each of which can be represented as a 2-D matrix. If a convolutional layer has N_{in} input channels, it will have N_{out} convolution filters that transform inputs into N_{out} output channels. In this equation 2.1, f_i represents the i^{th} input channel, and g_i represents the j^{th} output channel. The formula represents the computation for the output of a single neuron in a fully connected layer of a neural network without activation function applied to the output.

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

The $w_{i,j}$ term is a filter window used in convolutional operations, and its size is pre-determined by the designers. The values contained in $w_{i,j}$ and b_j are called weights and biases, respectively. k_h and k_w represent the filter width and height. To perform a forward calculation in

a convolutional layer, a total of $k_h \times k_w \times N_{in} \times N_{out}$ weights and N_{out} biases are required. When we consider the height and width of each output feature map to be g_h and g_w, respectively, we can estimate the total workload of the layer using Floating point operations (FLOPS).

$$\text{Workload (FLOPS)} = g_h \times g_w \times k_h \times k_w \times N_{\text{in}} \times N_{\text{out}} \times 2 \tag{2.2}$$

In the equation 2.2, the workload is multiplied by two because there are nearly the same amount of accumulation and multiplication operations. The values of g_w and g_h are typically determined based on the input size, whether the input is padded at the edges, and the stride of the sliding windows.

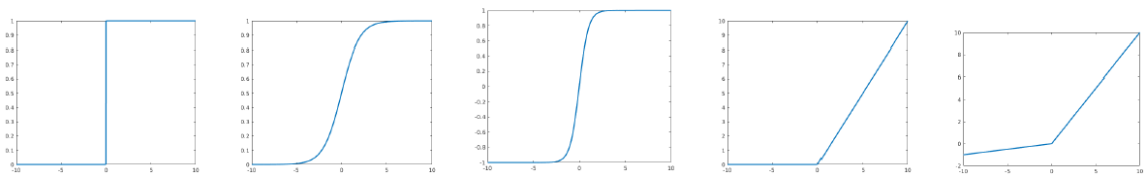
2.1.2 Other Typical Layer Types

In addition to convolutional layers, various types of layers have been introduced for different purposes such as creating non-linear models, compressing data, and generating more informative outputs.

- **Non-linearity layers**

  Non-linearity is crucial in deep learning architectures to provide the necessary non-linearity, which enables the model to learn complex relationships between inputs and outputs. The non-linear activation functions that are commonly used include binary step, sigmoid, tanh, ReLU, Leaky ReLU, ELU, and so on shown in figure2.1. The ReLU family of activation functions has become increasingly popular in recent times [11]. This type of function is piecewise linear, and it returns zero for negative inputs and the input value for positive inputs. Rectified Linear Unit (ReLU) has been demonstrated to be computationally efficient and outperforms other activation functions such as sigmoid and tanh in a range of deep learning applications, including image recognition and natural language processing.

  Leaky ReLU is used as the activation function in the convolutional layers of the YOLOv4-tiny neural network to prevent the "dying ReLU" problem and improve the flow of gradients during backpropagation [6]. Its use helps to maintain high accuracy while reducing the computational cost of the network, making it suitable for real-time object detection applications.



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

Figure 2.1: Non-linear activation function [12]

- **Batch normalization**

  It is a technique utilized to enhance the stability of a neural network [13]. Typically, input data to a neural network is uniform and may not be differentiable in all areas. This technique modifies the distribution of the data, resulting in quicker convergence and decreased overfitting. It can also address the internal covariate shift issue, where the input distribution to each layer changes as the parameters of the previous layers are updated during training, leading to slow convergence and poor performance. Batch normalization is applied after the convolutional layers and before the activation function.

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

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

  It takes g_j and $g_j^{'}$, which are the inputs and outputs of the layer, respectively. μ_B and σ_B^2 represent the mini-batch mean and variance, respectively. A small value denoted by ϵ is added to the denominator to avoid division by zero. The parameters ϵ are used to scale and shift the normalized values of g_j, and their values are learned during the training process.

- **Pooling layer**

  Max pooling is a valuable and effective method for decreasing the amount of data in deep learning models. It works by selecting a representative value to represent a small region, usually the maximum or average value within that region. This helps to reduce the amount of information in the data while preserving the essential features, making it simpler to work with and analyze the data. As a result, pooling is an efficient way to reduce the computational burden of deep learning models by downsampling the data.

- **Fully-connected layer**

  The fully-connected layer is utilized as the last layer in deep learning models to produce classification outcomes by taking into account all the features from previous layers. However, this layer increases computational demands because each neuron in the preceding layer is connected to every neuron in the fully-connected layer, leading to numerous connections and computations. Despite this challenge, fully-connected layers are still valuable for attaining precise classification outcomes in deep learning models.

- **Softmax layers**

  Softmax layers are frequently employed in image classification tasks because they can transform the raw scores assigned to each class into probabilities. Before softmax, class

classification is based solely on the scores assigned to each class, which can be vague and difficult to interpret, and the sum of these scores across all classes may not be a definite value. The softmax layer addresses these issues by converting the scores into probabilities using the below formulae which offers a more precise and easily understandable indication of the probability of each class.

$$P_i = \frac{e^{z|_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

where z_i is the input to the i-th neuron, K is the total number of neurons in the layer, and P_i is the output probability of the i-th neuron. In other words, the softmax function takes a set of scores (the z values) and squashes them into a set of probabilities that sum to 1. Each P_i represents the probability that the input belongs to class i. The denominator ensures that the sum of all the probabilities is 1, while the numerator exponentiates each score to ensure that the probabilities are non-negative.

2.1.3 Evaluation Metrics

Evaluation metrics for object detection are used to evaluate the performance of a model on an object detection task. These metrics provide a way to compare multiple detection systems fairly and objectively, either against each other or against a benchmark. Metrics based on average precision (AP) and its derivatives are used to assess the quality of object detections and benchmark them accordingly.

- **Intersection Over Union (Intersection over Union (IoU))**

  The fundamental principle underlying modern evaluation metrics for object detection is the Intersection-over-Union (IoU) overlap measure. This metric is defined as the ratio of the intersection of the detection bounding box and the ground truth bounding box to their union. By calculating the IoU, we can determine how well the predicted bounding box aligns with the ground truth bounding box. Typically, an IoU score greater than 0.5 is considered a good prediction. This threshold value may vary depending on the specific application or context.

- **True Positives** t_p

  In object detection tasks, true positives t_p refer to the number of detections with an IoU score greater than 0.5, indicating that the predicted bounding box overlaps sufficiently with the ground truth bounding box.

- **False Positives** f_p

  False positives f_p in object detection tasks indicate the number of predicted bounding boxes that either have a low degree of overlap with the ground truth bounding box (measured by an IoU score less than or equal to 0.5) or are identified as duplicates.

- **False negatives** f_n

  In object detection tasks, false negatives f_n represent the number of objects that are present in the ground truth but are not detected by the model, regardless of whether the undetected object has an IoU score greater than or less than 0.5. In object detection tasks, false negatives [FN] represent the number of objects that are present in the ground truth but are not detected by the model, regardless of whether the undetected object has an IoU score greater than or less than 0.5.

- **Precision**

  Precision is a measure that is utilized in object detection tasks to assess how accurately the model is predicting the bounding boxes. It is calculated by dividing the number of true positive detections by the sum of true positive and false positive detections. Essentially, precision evaluates the proportion of predicted bounding boxes that are accurate.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positives}}$$

- **F1 Score**

  The F1 score is a popular metric used in object detection to assess the overall performance of a model. It is calculated as the harmonic mean of precision and recall, which provides a balanced measurement of the model's capability to identify positive objects while minimizing false positives.

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

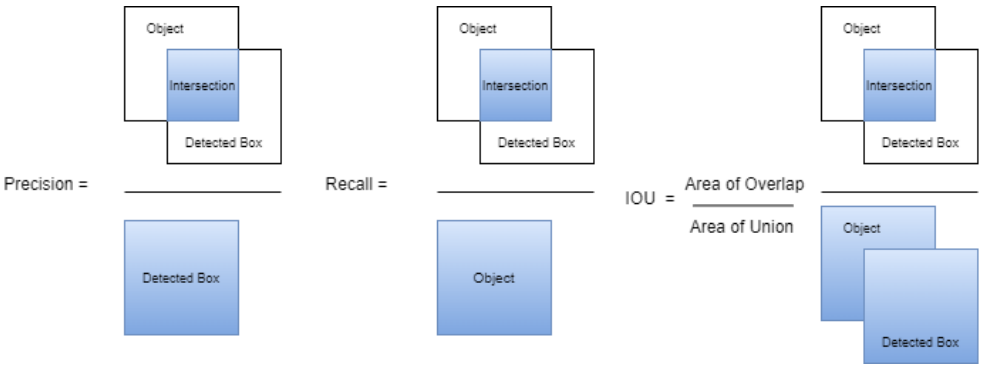


Figure 2.2: Precision vs Recall vs IoU [14]

- **Average Precision (AP)**

  To evaluate the performance of a detection model in object detection tasks, the precision-recall curve is often used, which displays the precision (y-axis) and recall (x-axis) values

calculated at different confidence thresholds used for detecting objects. The area under this curve, which is also known as the average precision (AP), provides a single numerical score that summarizes the overall performance of the detection model across all recall levels. The AP score measures how well the model can balance precision and recall across different confidence thresholds, with higher AP scores indicating better model performance. A perfect detection model would have an AP score of 1.0.

$$\text{Average Precision (AP)} = \int_0^1 P(r)dr$$

- **Mean Average Precision (mAP)**

  The Mean Average Precision (Mean Average Precision (mAP)) is a metric used in object detection tasks to evaluate the overall performance of a model across all object classes. It is the average of the AP values calculated for each class, with AP being the area under the precision-recall curve. The mAP@0.5 indicates that the calculation is done using an IoU threshold of 0.5 to determine whether a predicted bounding box overlaps with the ground truth box.

2.1.4 Dataset

The Commom Objects in Context (COCO) (Common Objects in Context) [11] dataset is a comprehensive collection of images that are used extensively in computer vision research for object detection, segmentation, and captioning. With over 330,000 images and 2.5 million object instances, it is considered a large-scale image recognition dataset. The dataset features images from various categories, including people, animals, vehicles, household items, and outdoor scenes. This diversity makes it a valuable resource for researchers working on computer vision problems related to real-world scenarios.

The COCO dataset is an ideal choice for training and assessing the YOLOv4-tiny model for several reasons.

- Firstly, the dataset offers a wide range of images that cover various object categories and situations. This enables the model to learn and identify various objects, making it helpful for various applications.

- Secondly, the COCO dataset contains comprehensive annotations for each image, including the category, location, and size of each object. This allows the YOLOv4-tiny model to identify and locate objects within an image accurately and classify them correctly, even in complex scenes with multiple objects and occlusions.

- Thirdly, the COCO dataset is widely recognized in the computer vision community, making it a standard benchmark for object detection and segmentation models. This enables the YOLOv4-tiny model's performance on the COCO dataset to be compared with other state-of-the-art models and assessed in a consistent and unbiased manner.

2.1.5 Frameworks

Deep neural network (DNN) frameworks are software tools that provide pre-built implementations of common deep-learning algorithms. Some frameworks even come with pre-trained neural network models. These tools help speed up the development and research in the field of deep learning. Frameworks work with a higher level of abstraction that enables users to define the structure of the application. Configuration files contain the application skeleton that specifies the layer types, number of neurons per layer, input data shape, and other details. Many frameworks can also use GPUs to accelerate the inference and learning process.

Although YOLO was initially implemented in the Darknet framework, it has also been implemented in other popular frameworks. Choosing the right framework is essential to solving the problem effectively. In addition to Darknet, this section will discuss two other widely-used frameworks.

- **Darknet**: Darknet is a neural network framework that is free and available for anyone to use [15] It is created using the programming languages C and CUDA and was developed by Joseph Redmon. The framework has gained recognition because it implemented YOLO , which is an object detection algorithm known for its high accuracy and real-time object detection capabilities. The framework can be utilized with both CPU and GPU acceleration and is supported by various platforms such as Linux, Windows, and macOS.
  The most significant benefit of the darkent is its speed and efficiency. Its lightweight design and optimized algorithms allow it to execute fast inference times even on low-powered devices. Moreover, the framework has an intuitive and straightforward API, which makes it simple for researchers and developers to create, train, and evaluate their models.

- **Caffe**: Caffe is a framework for deep learning that was developed by the Berkeley Vision and Learning Center [16]. It is created with a focus on being efficient, modular, and extensible. Caffe supports both CPU and GPU acceleration, similar to Darknet, and is written in C++ and CUDA, with bindings for other programming languages such as Python and MATLAB.
  Caffe is particularly strong because of its pre-trained model zoo. The model zoo contains a wide range of pre-trained models for various computer vision tasks such as object detection, segmentation, and classification. This feature enables developers and researchers to quickly build and evaluate their models for their specific use cases without needing to train their models from scratch.

The framework's modular design is another advantage of Caffe, which allows customization and extension of the framework. This modular structure makes Caffe an ideal platform for the research and development of new deep-learning algorithms and architectures.

- **Tensorflow** TensorFlow is a deep learning framework created by the Google Brain team that has become widely used and is considered one of the most popular open-source deep learning frameworks available [17]. TensorFlow works on both CPU and GPU and can be used on various platforms such as Linux, Windows, and macOS.
  One of the significant advantages of TensorFlow is its ability to be versatile and scalable, allowing users to train various types of deep learning models such as convolutional neural networks, recurrent neural networks, and generative adversarial networks. TensorFlow is also optimized for distributed computing, making it possible to train large-scale models or on distributed clusters.
  Another benefit of TensorFlow is its vast community of users and contributors, which provides numerous resources, including pre-trained models, tutorials, and extensions. This enables researchers and developers to quickly build and evaluate models for their specific applications.

Why Darknet over Tensorflow or Caffe Framework.?

Darknet is a preferred framework over Caffe and TensorFlow for the YOLOv4-tiny model for several reasons. One of the significant advantages of Darknet is its speed and efficiency, as it is optimized for object detection tasks and written in C and CUDA, making it fast and memory efficient, particularly when running on GPU. Additionally, Darknet's open-source nature enables developers to customize and modify it to suit specific requirements, which is crucial for improving the performance of the YOLOv4-tiny model. Darknet was built specifically for the YOLO family of object detection models, including YOLOv4-tiny, and is optimized for their unique architecture and requirements. Darknet also provides a wide range of tools and functions for training, testing, and evaluating object detection models, including data augmentation, model visualization, and performance evaluation metrics, making it a comprehensive and integrated solution for object detection tasks.

2.2 YOLO: Real-Time object detection algorithm

Joseph Redmon et al. introduced the first generation of You Only Look Once (YOLO) in 2016, which uses a single Convolutional Neural Network (CNN) to predict both class probabilities and bounding boxes [5]. YOLO divides the input image into grids of size S x S, with each grid capable of providing B bounding boxes, along with confidence values for the boxes and probabilities for C classes. YOLO's low latency and high throughput are the key factors behind its state-of-the-art performance in real-time detection.

In 2017, YOLOv2, an improved version of YOLO, was launched with a range of upgrades such as batch normalization and *anchor boxes* [18]. Anchor boxes refer to preset bounding boxes that have fixed scales between width and height. By using anchor boxes instead of a fully-connected layer, the size of objects that can be detected is limited, but it enhances the recall rate and separates object classifications from bounding boxes. To fulfill the requirement of a faster and more efficient object detection network that can run in real-time on devices with limited computational resources like mobile phones or embedded systems, the YOLOv2 tiny model was developed. Compared to the full YOLOv2 network, the tiny model has fewer layers and parameters, which results in faster processing times while maintaining a satisfactory level of accuracy of about 57% in object detection tasks. However, the anchor boxes used in YOLOv2-tiny restrict the range of object sizes that can be detected. This can result in missed detections or inaccurate bounding box sizes for objects outside of the preset range. Moreover, The YOLOv2-tiny model was designed to detect a limited number of object classes, typically around 20. This can be a limitation in scenarios where a larger number of object classes need to be detected.

In 2018, a newer version of YOLO called YOLOv3 was introduced, which improved the accuracy of the model [19]. Instead of using softmax activation at the output of the network, sigmoid activation was used to address the poor performance of softmax when multiple labels correspond to the same bounding box. YOLOv3 also uses a structure similar to feature pyramid networks, combining upsampled features with those from previous layers to process the combined features further through several convolutional layers. This structure makes predictions across different scales easier, leading to improved accuracy.

The YOLOv3 model has a workload of 65.86 GFLOPs and achieves a 51.5% mAP@0.5 on the COCO test-dev dataset while running at 35 FPS on a Pascal Titan X GPU. For resource-constrained environments, a lighter model called YOLOv3-tiny is available. This model is more suitable for embedded applications and can achieve a 33.1% mAP@0.5 on the same test dataset but with a much higher frame rate of 220 FPS on the same GPU device.

In 2020, YOLOv4 was introduced as an upgraded version of previous YOLO models, offering faster, more accurate, and more robust object detection capabilities. One of the key advancements of YOLOv4 is its utilization of a "bag of freebies" and a "bag of specials" [20]." The former involves standard techniques like data augmentation, label smoothing, and random training shapes to enhance model performance, while the latter comprises more sophisticated techniques such as the Mish activation function, self-adversarial training, and DropBlock regularization. YOLOv4 also incorporates a CSPDarknet53 backbone, which is a modified version of the Darknet architecture that includes cross-stage partial connections to improve information flow within the network. Along with a spatial pyramid pooling layer and a path aggregation network, the CSPDarknet53 backbone enhances the model's ability to detect objects at varying scales

YOLOv4-tiny is a less resource-intensive version of the YOLOv4 model used for object detection [20]. Its design is suitable for deployment in limited-resource environments like mobile devices and embedded systems. Compared to the full YOLOv4 model, YOLOv4-tiny has a smaller model size and fewer layers, enabling it to operate at a faster speed while utilizing less memory. Despite its reduced size, YOLOv4-tiny still attains high accuracy, with a mean average precision (mAP) of 43.5% on the COCO dataset.

Similar to YOLOv3-tiny, YOLOv4-tiny utilizes anchor boxes for bounding box prediction and a feature pyramid network for detecting objects at different scales. It incorporates some of the enhancements from YOLOv4, including the use of CSPDarknet53 as the backbone network and the integration of Spatial Pyramid Pooling (SPP) (Spatial Pyramid Pooling) to capture context information. The implementation of a new architecture in the Backbone which is called CSPDarkent 53 and the modifications in the Neck have improved the mAP(mean Average Precision) by 10%.

2.3 Motivation For choosing YOLOv4-tiny

Table 2.1 gives the comparison of the YOLO model for object detection based on the trade-off between accuracy (mAP) and computational complexity (GOPS) [20] [19] [18]. The Original YOLO models have more than 8x parameters as compared to their tiny counterparts. The YOLO Tiny models are more desirable than the original YOLO models due to their smaller size, faster speed, and lower computational requirements. They are particularly suitable for cases that necessitate real-time object detection on devices with limited power, like embedded systems, drones, and smartphones. Moreover, the YOLO Tiny models usually outperform the original YOLO models when detecting small or distant objects since they are trained on smaller input images, making them more adept at identifying such objects.

Model	Parameters	mAP	GOPS
YOLOv2	50.80M	76.80%	29.46
YOLOv2-tiny	6.97M	57.10%	5.46
YOLOv3	62.90M	57.90%	65.34
YOLOv3-tiny	8.97M	33.10%	5.40
YOLOv4	63.80M	46.70%	62.00
YOLOv4-tiny	6.90M	44.00%	6.62

Table 2.1: Comparison of YOLO models.

2.3.1 Theorectical upper bound performance:

Before selecting the correct version of the YOLO algorithm to be implemented on a target platform, it is important to estimate a theoretical upper-bound performance using the number of computational resources like Multiply-Accumulate (MACC) units present in the target platform. The target platform chosen for this project is ZedBoard. In CNN, convolutional

operations are the most computationally intensive operations that are done in a repetitive manner for an entire layer in CNN which takes about 99% [8]. Convolution operation requires Multiply-Accumulate (Multiply and Accumulate (MACC)) operation which is mapped to DSP slices present on the target platform. Therefore for the analysis, 1 DSP slice will be considered as a processing element (Processing Element (PE)) which is designed to perform repetitive multiplication and addition operations, making them well-suited for convolution operations in deep-learning models.

The Theoretical upper bound performance is given by Frames per second which relates to the throughput of the design and the processing elements present in the target platform as follows :

$$\text{Frame per seconds (Frame Per Seconds (FPS))} = \frac{\text{(Throughput of 1 PE x Number of PEs)}}{\text{Workload}}$$

where:

- Throughput of 1 PE is the number of MACC operations that can be performed by 1 PE per second.
- Number of PEs is the total number of PEs in the target platform.
- Workload is the number of MAC operations required for one inference given as Giga Operations per second (GOPS).

In ZedBoard, there are about 220 DSP slices and each Slice is of size 18x25 fixed-point multiplications. Assuming each DSP slice can perform 1 MAC operation per cycle, Therefore the throughput of a slice per cycle is given as

$$\text{Throughput of 1 PE per Cycle} = 1 \text{ MAC}$$

Assuming all PEs are active at the same time i.e. all 220 DSP slices are active at the same time. Additionally, if the clock frequency of the design is 100 MHz. Then

$$\textbf{Throughput of 1 PE} \times \textbf{Number of PE} = 1 \times 100 \times 10^6 \times 220 = 22 \text{ GOPS/secs}$$

The workload for each tiny model can be used from table 2.1.

Yolo version	**Computations (GOPS)**	**Theoretical Upper bound FPS**
Yolov2	29.46	0.74
Yolov2 tiny	5.56	3.95
Yolov3	65.34	0.33
Yolov3 tiny	5.4	4.07
Yolov4	52.00	0.42
Yolov4 tiny	6.62	3.33

Table 2.2: Comparision of FPS

Table 2.2 gives the FPS results for the YOLO versions shown. The reason for not choosing the yolov3-tiny model was its low accuracy of approximately 33% mAP. Selection between YOLOv2-tiny and YOLOv4-tiny depends on various factors, including the balance between accuracy and speed and the robustness to variation in input data. In terms of speed, which is crucial, YOLOv4-tiny appears to be a better option as it has fewer layers to process as compared to YOLOV2-tiny which means that it has fewer layers and parameters. While this makes the network faster and more efficient, it also reduces its capacity to learn complex features and patterns in the input data, which can affect its accuracy which explains why YOLOv4-tiny has a lower 44% mAP as compared to YOLOv2-tiny (57%). This lower accuracy is also due to the fact that the YOLOv4-tiny model is trained for detecting 80 different classes as compared to 20 classes in the case of YOLOv2-tiny but this also means that YOLOv4-tiny is more robust for new input. Furthermore, YOLOv4-tiny is a newer model that includes various improvements and optimizations compared to the earlier model, such as a better backbone architecture and an enhanced feature pyramid network, which contributes to its accuracy and efficiency. These are some of the reasons why YOLOv4-tiny is preferred over other tiny versions for object detection algorithms.

2.4 YOLOv4-tiny

YOLOv4-tiny has a two-part structure that includes detection layers and a backbone network. The backbone network is a modified version of CSPDarknet53, which is a variant of the Darknet network that uses cross-stage partial connections to enhance feature propagation. The CSPDarknet53-tiny architecture implemented in YOLOv4-tiny has a smaller depth and width than the original CSPDarknet53, making it a smaller and faster network.

- **CSPDarknet53-tiny** The backbone network in YOLOv4-tiny, as shown in Figure 2.3 is constructed using CSPDarknet53-tiny, a simplified version of the CSPDarknet53 architecture used in the larger YOLOv4 model. The abbreviation CSP stands for Cross Stage Partial connections, which link the layers in the network. The CSP technique splits the layers into two branches, with one branch performing the convolutional operation and the other performing identity mapping. The outputs of these branches are then combined through concatenation before passing to the next layer, improving gradient flow and reducing overfitting.

  The CSPDarknet53-tiny architecture includes three convolutional layers and three CSP-Block modules. The first two convolutional layers consist of a convolutional layer and a Leaky-ReLU layer, with each convolutional layer having a kernel size of 3 and a stride of 2. The final convolutional layer has a kernel size of 3 and a stride of 1. In contrast to the Mish activation function used in the original CSPDarknet53 architecture, CSPDarknet53-tiny uses the simpler Leaky-ReLU activation function to simplify the calculation process.

The YOLOv4-tiny model employs two feature maps with different sizes, namely 13 x 13 and 26 x 26, for detecting objects. For each feature map, three anchor boxes of varying sizes are utilized to predict three bounding boxes. Prior to training, k-means clustering is applied to determine the number of anchor boxes, resulting in a total of six predefined boxes across the three scales.

To handle the issue of multiple overlapping detection bounding boxes, YOLOv4-tiny uses a variation of Non-Maximum Suppression (NMS) called distance intersection over union non-maximum suppression (DIoU-NMS). This approach takes into account the distances between the center points of the bounding boxes, especially when they do not overlap. The threshold for DIoU-NMS in this case is set to 0.4.

The YOLOv4-tiny architecture consists of 21 convolutional layers out of which 19 are depthwise convolutions followed by LeakyRelu activations functions and 2 are fully connected layers. There are about 3 Maxpool Layers, 11 route layers, 1 Upsample layer, and 2 Yolo Layers for detecting images at different scales of 26x26 and 13x13 respectively. The detailed block diagram and topological feature of each layer are presented in figure 2.4. Table 2.4 gives the topological configurations on each layer with Billion Floating point operation (BFLOPS) per layer.

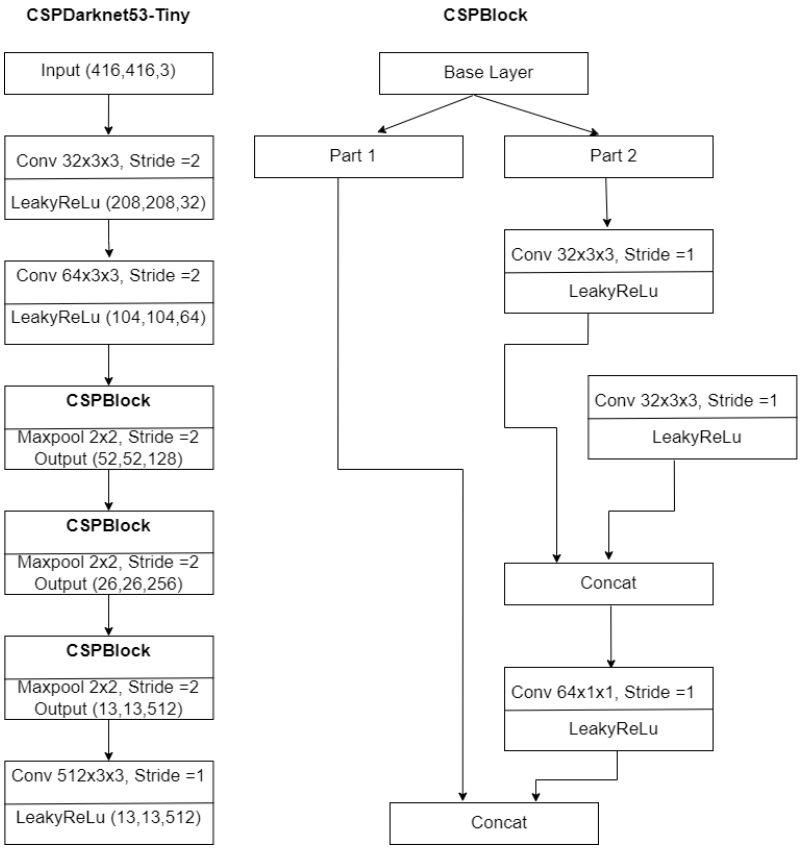


Figure 2.3: YOLOv4-tiny Architecture

Table 2.3: Network architecture of YOLOv4-tiny

Layer	Filters	Size	Input	Output	BFLOPS
conv	32	3 x 3 / 2	416 x 416 x 3	208 x 208 x 32	0.075
conv	64	3 x 3 / 2	208 x 208 x 32	104 x 104 x 64	0.399
conv	64	3 x 3 / 1	104 x 104 x 64	104 x 104 x 64	0.397
route	2 *(output from layer 1/2)*				
conv	32	3 x 3 / 1	104 x 104 x 32	104 x 104 x 32	0.199
conv	32	3 x 3 / 1	104 x 104 x 32	104 x 104 x 32	0.199
route	5 4 *(outputs from layers 5 and 4)*				
conv	64	1 x 1 / 1	104 x 104 x 64	104 x 104 x 64	0.089
route	2 7 *(outputs from layers 2 and 7)*				
max	2 x 2 / 2	104 x 104 x 128	52 x 52 x 128	26 x 26 x 128	-
conv	128	3 x 3 / 1	52 x 52 x 128	52 x 52 x 128	0.397
route	10 *(output from layer 1/2)*				
conv	64	3 x 3 / 1	52 x 52 x 64	52 x 52 x 64	0.199
conv	64	3 x 3 / 1	52 x 52 x 64	52 x 52 x 64	0.199
route	13 12 *(outputs from layers 13 and 12)*				
conv	128	1 x 1 / 1	52 x 52 x 128	52 x 52 x 128	0.089
route	10 15 *(outputs from layers 10 and 15)*				
max	2 x 2 / 2	52 x 52 x 256	26 x 26 x 256	13 x 13 x 256	-
conv	256	3 x 3 / 1	13 x 13 x 256	13 x 13 x 256	0.397
route	18 *(output from layer 1/2)*				
Conv	128	3 x 3 / 1	26 x 26 x 128	26 x 26 x 128	0.199
Conv	128	3 x 3 / 1	26 x 26 x 128	26 x 26 x 128	0.199
Route	21, 20	-	-	26 x 26 x 256	-
Conv	256	1 x 1 / 1	26 x 26 x 256	26 x 26 x 256	0.089
Route	18, 23	-	-	26 x 26 x 512	-
Max	-	2 x 2 / 2	26 x 26 x 512	13 x 13 x 512	-
Conv	512	3 x 3 / 1	13 x 13 x 512	13 x 13 x 512	0.397
Conv	256	1 x 1 / 1	13 x 13 x 512	13 x 13 x 256	0.044
Conv	512	3 x 3 / 1	13 x 13 x 256	13 x 13 x 512	0.399
Conv	255	1 x 1 / 1	13 x 13 x 512	13 x 13 x 255	0.044
YOLO	-	-	-	13 x 13 x 255	-
Route	27	-	-	13 x 13 x 256	-
Conv	128	1 x 1 / 1	13 x 13 x 256	13 x 13 x 128	0.011
Upsample	-	2x	13 x 13 x 128	26 x 26 x 128	-
Route	33, 23	-	-	26 x 26 x 384	-
Conv	256	3 x 3 / 1	26 x 26 x 384	26 x 26 x 256	1.196
Conv	255	1 x 1 / 1	26 x 26 x 256	26 x 26 x 255	0.088
YOLO	-	-	-	26 x 26 x 255	-

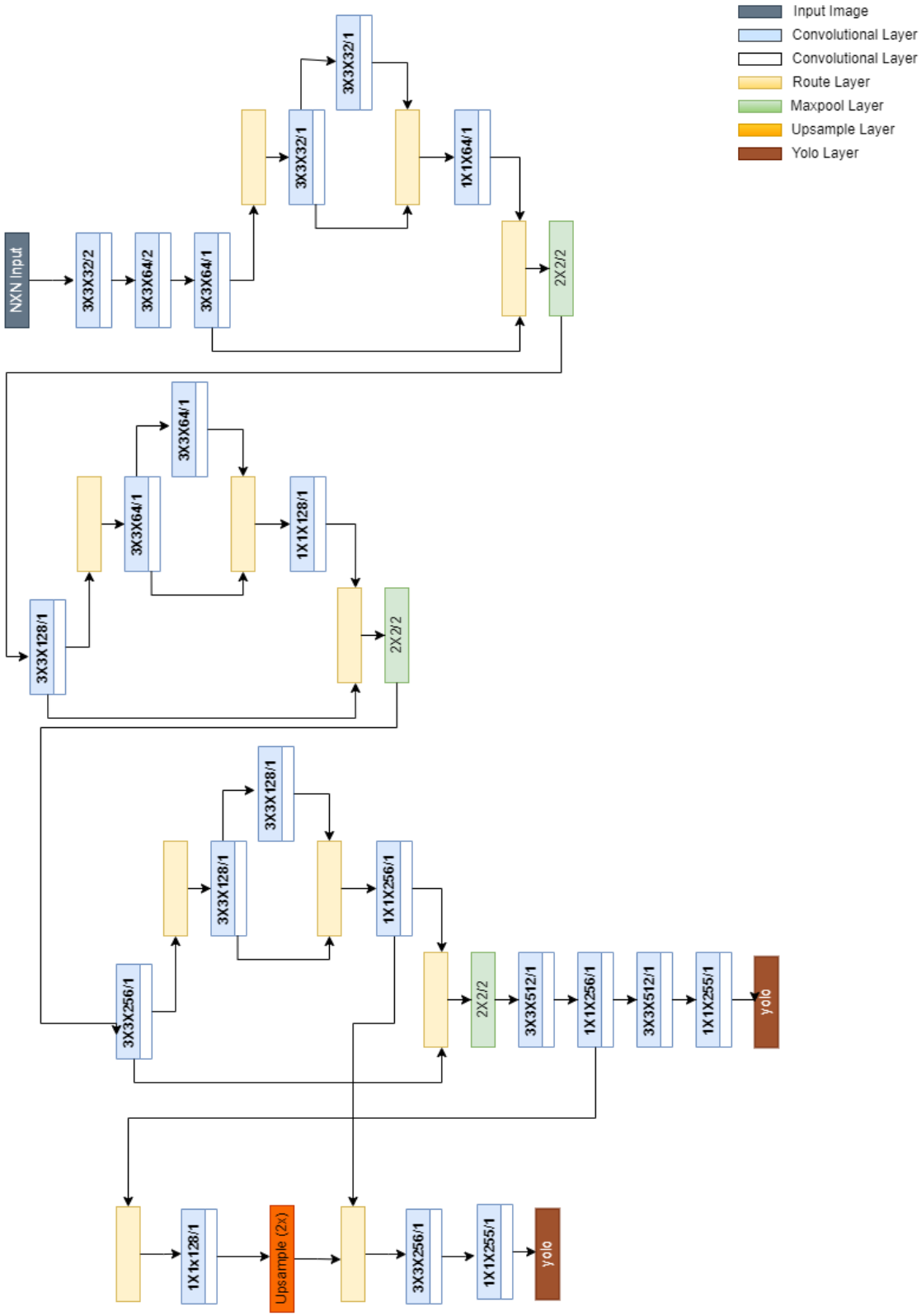


Figure 2.4: YOLOv4-tiny Architecture

Chapter 3

Literature Review

This chapter focuses on prior research that involves implementing Convolutional Neural Networks (CNNs) on FPGAs, as well as the specialized techniques utilized in embedded applications. The major obstacle is the significant computation and memory storage demanded by CNNs.

3.1 2-D Convolution on FPGA

Numerous research studies are currently focused on optimizing 2-D convolution for embedded FPGA devices, with a primary focus on enhancing memory storage, data bandwidth, and parallelism. By achieving optimal performance in these areas, FPGA devices can handle the computational demands of convolutional neural networks (CNNs) more efficiently than traditional processors, enabling high-performance computing with low power consumption.

The two primary buffer schemes used in FPGA-based convolutional neural networks (CNNs) are full buffering (FB) and partial buffering (PB) [21]. Full buffering involves accessing the entire feature map in a sequential line-by-line manner, ensuring that every input pixel is read from external memories only once. This data is then stored locally and reused by the sliding window. However, this approach typically requires a large on-chip memory on FPGA due to the need to store all input data locally.

In contrast, partial buffering divides the feature map into multiple regions and loads only one region at a time to the FPGA device. Although this approach requires less on-chip memory compared to full buffering, it has the drawback of loading the same data multiple times, which increases the frequency of accessing external memory and demands higher data bandwidth.

An alternative approach to partial buffering is the use of the Image to Column (Im2Col) algorithm [22] [23], which involves splitting the input feature map into batches that have the same size as weight windows. This transformation allows the convolution operation to be represented as a matrix multiplication, which can be further expanded into 1-D vectors. In this way, convolution can be computed as the dot product between the input vector and the

weight vector.

3.2 Fixed-point Quantization

Most FPGAs do not have floating-point hard cores, which makes it challenging to handle the high precision required for neural network training. A common approach is to train the network using floating-point representations on a GPU and then convert it to a lower-precision version with or without retraining. This technique has been documented in research papers [24] [25].

Fixed-point implementation requires determining the number of bits allocated for the integer and fraction parts. In linear quantization, the bit width of integers depends on extreme values and the possibility of overflow. The length of the fraction part affects quantization error, and the step size of quantization shapes the data distribution. Therefore, it is crucial to investigate the trade-off between bit-width and network precision for different network types. This analysis is necessary for accurate implementation of neural networks on FPGAs.

Qiu et al. [26] introduced a dynamic strategy in 2016 that selects different fraction lengths for each layer in a neural network. Re-quantization is then performed between layers by shifting bits in the activation. This dynamic quantization approach is more efficient in terms of resource usage, especially when there is significant variation in weight and data distributions between layers.

In addition to linear quantization, there are alternative methods to reduce bit-width even further. Logarithmic data representation is a technique that transforms weight and activation values into the log domain. This approach can achieve compression because the original distributions of the data are typically non-uniform [27].

Rastegari et al. [28] introduced Binary-Weight-Networks and XNOR-Networks that binarize either the weights or both the weights and data. However, most studies on binarized networks concentrate on image classification rather than object detection due to the significant loss of precision that binarization causes, particularly in bounding box predictions.

3.3 Dynamically Configurable Architecture

Different layers within a neural network can have different typology parameters, such as the number of neurons, activation functions, and weights. These parameters can affect the computational requirements and memory access patterns of each layer, making it challenging to optimize performance for the entire network.

One possible solution to this challenge is to design specific hardware for each layer of the network. By customizing the hardware for each layer, it is possible to reduce off-chip memory

transactions and improve performance. For example, hardware accelerators can be designed to perform the specific computations required by each layer, which can lead to faster and more efficient processing [2]. However, one disadvantage of this approach is that it can require significant resources to put the entire network on hardware.

A potential solution to address the compatibility issue and huge storage requirements of intermediate results is to develop a unified hardware architecture that can accommodate all layers. This approach involves controlling multiplexers within the architecture to support a range of convolution types and configurations. This strategy has gained popularity in FPGA-based CNN implementations, with Chakradhar et al. [2] proposing it in 2010. However, a drawback of this approach is that it may sacrifice some performance for the sake of compatibility.

3.4 Related Work on YOLO

The Eyeriss architecture, introduced in [29], can minimize off-chip memory access of any CNN shape using a spatial architecture that includes 168 processing elements. This architecture employs a computational scheme named "row stationary" which enhances parallelism to achieve high throughput while optimizing the data movement to improve energy efficiency. The authors evaluated the Eyeriss architecture's performance on Xilinx V707 FPGA, which has 2800 DSP slices, using two popular CNNs: Alexnet and VGG-16. With 16-bit fixed-point arithmetic precision, Eyeriss achieves a frame rate of 37.1 frames/sec and throughput of 23.1 with a DRAM access of 0.0029 per MAC operation for Alexnet CNN, while for VGG-16, the frame rate is 0.7 frames/s with a DRAM access of 0.0035 per MAC operation.

In a recent paper [30], researchers proposed a fully scalable and configurable IP core to improve parallelism during both inference and training time. The core is responsible for accelerating the execution of all the algorithm steps, including pre-processing and model inference, and has been configured for real-time execution of YOLOv3-Tiny and YOLOv4-Tiny models. This core has been integrated into a RISC-V-based system-on-chip architecture and prototyped in an UltraScale XCKU040 FPGA, resulting in a complete system called Soc-Yolo. The IP core consists of a matrix of vector functional units that perform MAC computations, exploiting parallelism with both input and output Feature Map(FM) such as Intra-convolution (concurrently computing multiplications within 2D convolutions), Inter-convolutions (Multiple 2D convolutions computed concurrently), and Intra-FM (computing multiple pixels of a single output feature map), inter-FM (concurrently processing multiple output feature map) and enhancing pixel and weight sharing. In performance tests, the YOLOv3-Tiny detector implemented in the SoC-YOLO platform achieved a frame rate of 32 FPS, with a peak throughput of 238 GOPS and an execution time of 30.9 ms. Similarly, the YOLOv4-Tiny detector on the SoC-YOLO platform achieved a frame rate of 31 FPS, with a peak throughput of 357 GOPS and an execution time of 32.1 ms. These results show that the Soc-Yolo-based implementation of the YOLOv3-Tiny and YOLOv4-Tiny models is

significantly faster than the CPU version, with speed-ups of nearly 27x and 33x, respectively.

A parameterized FPGA-tailored architecture is proposed in [1] for low-latency detection with tiny YOLO3. The presented work is based on low-latency object detection on a low-end FPGA device like Zedboard. The paper also carries out a design space exploration for identifying design points that optimize latency while meeting the resource constraints of the FPGA device. The developed accelerator is utilized for the execution of all layers of the network through a run-time parameter setting, which is also parameterizable at compile time using Vivado HLS. The architecture is 1.5x more power efficient compared to the floating-point YOLOv3-tiny implementation on a Pascal Titan X and is 290x faster compared to the fully Arm Cortex A9 implementation. The proposed architecture achieves a frame rate of 1.88 FPS and 10.45 GOPs throughput with the low-cost FPGA evaluation board. Moreover, the proposed architecture sees a 290x improvement in latency compared to the hardcore processor of the device, achieving at the same time a reduction in mAP of 2.5 pp (30.9% vs 33.4%) compared to the original model under 16-bit fixed-point implementation.

In [31], a method is proposed to accelerate the YOLOv4 algorithm using FPGA in order to minimize inference time while maintaining high accuracy. The researchers evaluated resource utilization and inference time as important parameters for hardware acceleration of object detection algorithms. The proposed accelerator was designed for the Zynq-7000 SoC to maximize resource utilization and achieve large throughput. They used Vivado 2020.1 to prototype the hardware acceleration of the YOLOv4 algorithm, and employed loop pipelining optimization and loop tiling approach to enhance system throughput and reduce memory access, respectively. The proposed implementation achieved better resource utilization with 23.2k LUTs, 45.8 flip flops, 115 BRAMs, and 174 DSPs under the optimum frequency of 100 MHz. The design claimed to achieve a peak throughput of 189.14 GOP/s and 30 fps performance with an effective resource utilization of about 80%.

In [32], the authors describe an FPGA implementation of the lightweight YOLOv2 algorithm using a combination of binarized CNN and parallel State Vector regression for bounding box prediction and class estimation. The proposed architecture was implemented on the Xilinx Inc. Zynq UltraScale+ MPSoC zcu102 evaluation board. The binarized CNN was used for classification, while the parallel SVR was used for both classification and localization. The proposed architecture achieved an inference time of 24.5 msec per image, which corresponds to a frame rate of 40.81 FPS with dynamic board power consumption. Compared to the ARM Cortex-A57, the proposed architecture was 177.4 times faster and 1.1 times more power-hungry, with a performance per power efficiency that was 158.9 times better. The performance per power efficiency was calculated to be 9.06 FPS/W with a power consumption of 4.5 W.

In [33], an improved architecture is proposed for deep learning on the Tiny-Yolo network using Xilinx's high-performance ZYNQ SoC FPGA chip. The proposed architecture optimizes

the YOLO network model, replaces the activation function, and uses a 16-bit fixed-point on the weight parameters to efficiently utilize the resources present in the FPGA. The proposed architecture achieves an inference time of 51.9 ms or an FPS of 19.26, which is 44.9 times faster than the CPU. However, the accuracy is slightly lower, which is about 48.5% mAP compared to 51.3% mAP in the CPU.

The article [34] proposes a low-cost, hardware-efficient architecture for real-time object detection of YOLOv2-tiny on a Xilinx XC7Z035 FPGA. They used quantization-aware training to decrease precision while preserving accuracy. They utilized an open-source tool named ZigZag for design space exploration to optimize latency and reduce memory access. The proposed architecture employed two metrics, $\text{DSP}_{\text{efficiency}}$ and $\text{Cost}_{\text{efficiency}}$, to determine the optimal PE array size, spatial/temporal unrolling, and memory hierarchy with respect to latency. The design achieves an FPS of 11.54 and 23.07 for operating frequencies of 100MHz and 200 MHz, respectively. The resulting hardware is highly efficient with a $\text{DSP}_{\text{efficiency}}$ of 90% and a $\text{Cost}_{\text{efficiency}}$ of 0.146 GOPS.

New Accelerator for DCNN is proposed in [35] which leverages all forms of parallelism i.e. inter-Layer, inter-output, inter-kernel, and intra-kernel to minimize the execution time. An improved tiling strategy is adopted inside the convolutional kernel to improve the performance. Analytical modeling for the proposed architecture with respect to the Alexnet architecture is carried out to find the best design parameters for implementing Xilinx Virtex7 FPFA. The proposed architecture achieves a throughput of 84.2 GOPs with 32-bit float precision.

Chapter 4

Relevant Tool: Vitis Unified Software Platform & Vivado HLS

4.1 WorkFlow Used For Creating Accelerated Application

The Vitis Unified Software Platform is a powerful development environment that enables developers to build accelerated applications on heterogeneous computing systems. The platform is developed by Xilinx, a leading provider of FPGA and other hardware solutions for accelerating compute-intensive workloads. The Vitis platform provides a suite of tools, libraries, and APIs that abstract the complexities of hardware programming, making it easier for developers to build high-performance, energy-efficient applications. The platform supports a range of programming languages, including C, C++, OpenCL, and Python, and provides a high-level programming model that simplifies the development process.

Vivado is used under the hood in the Vitis Unified Software Platform. Vivado is used to create the hardware design for the FPGA, which can then be programmed and accelerated using the Vitis platform. The Vitis platform provides software tools and libraries that enable developers to build, test, and optimize their applications on the FPGA design created in Vivado. So, Vivado and Vitis Unified Software Platform work together to provide a complete solution for building accelerated applications on Xilinx devices. The workflow presented in Figure 4.1 is followed in this thesis for creating an accelerated application using the Vitis Unified software platform.

The design process with High Level Synthesis (HLS) involves multiple stages and tools. First, designers develop and test the design specification using C, and C++. Next, HLS is used with technology and clock settings to generate Register Transfer Level (RTL) code. The resulting RTL code and IP blocks are then integrated using the Vivado design suite's IP Integrator. Finally, an application is built on this FPGA design using the Vitis Unified Software Platform, which facilitates hardware-software codesign.

In this chapter, this chapter will discuss the process of creating a compatible HLS C++ design for Vivado HLS. Firstly, section 4.2 discusses about both built-in datatypes and custom datatypes used in creating a C++ design.

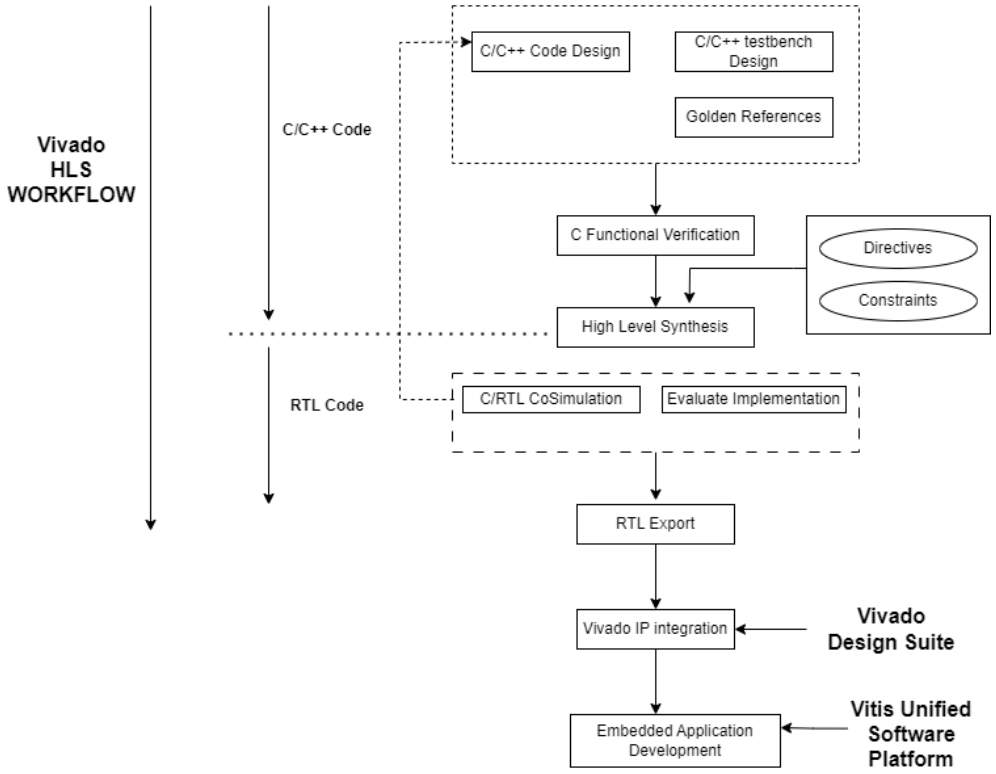


Figure 4.1: Work flow for creating accelerated application

4.2 Data types

In High-level synthesis, data types are used to define the type and size of the variables used in the design. As a high-level programming language is used to describe hardware, more accurate descriptions of (sub-byte) data types are required. Vivado allows the implementation of different number formats, including integers, fixed-point data types, and floating-point data types. All these concepts are described below.

- `ap_(u)fixed<W,I,Q,O,N>`
  This data type represents a signed or unsigned fixed-point number. It is defined by specifying the total number of bits and the number of integer bits. The data type can be defined after including `ap_fixed.h` or `ap_ufixed.h` header file.
  Here, W is the total width of the fixed-point number in bits, I is the number of integer bits in the fixed-point number to the left of the binary point, Q refers to quantization mode in the fixed-point number, O is the overflow mode, and N is the number of saturation bits in various overflow modes. It is not necessary to give all the parameters to the `ap_fixed` for representing fixed-point values as shown in figure 4.1. There are various

quantization modes and overflow modes which impact the precision of the representable values [36]. `AP_RND_CONV` has been used as a quantization mode for designing IP blocks as it provides convergent rounding, which ensures more accurate results by rounding to the nearest representable value with ties rounded to the nearest even value. `AP_SAT` has been used in designing IP blocks as it enables saturation behavior, which prevents overflow or underflow issues. When a result exceeds the valid range, it is clamped to the maximum or minimum representable value, ensuring data integrity and avoiding data loss.

```
#include <ap_fixed.h>
ap_fixed<16, 8> weight_type;
ap_fixed<16, 8> data_type;
ap_fixed<32, 16> acc_type;
ap_fixed<16,8, AP_RND_CONV, AP_SAT> x;
```

Listing 4.1: Fixed-point data types

- `ap_(u)int<N>`
  This data type used in Vivado HLS represents a fixed-width signed and unsigned integer. It allows users to define the number of bits they need for their specific design by specifying the number of bits in the angle brackets. The 'ap' in the data type refers to arbitrary precision, which means that the bit-width can be customized. For instance, listing 4.2 declares two datatypes of type ap_uint of length 16 bits and 32 bits and initializes them to a value within the limits representable by the number of bits.

```
#include <ap_int.h>
ap_int<8> a = 50;
ap_uint<16> x = 65535;
ap_uint<32> y = 4294967295;
```

Listing 4.2: Arbitary precision integer data types

4.3 HLS Stream Library

The HLS stream library is primarily designed in High-Level Synthesis (HLS) tools to model and simulate streaming data interfaces within the design. The HLS stream library is synthesizable. The class `hls::stream` creates buffers that behave like first-in, first-out (FIFO) queues for data transmission between various hardware modules within an FPGA design. There is no requirement to define the size of an `hls::stream`. They are read from and written sequentially. That is, after data is read from an `hls::stream`, it cannot be reread. This section will discuss such data transfers between IPs and data can be read or written from this stream.

4.3.1 Using HLS stream

To use `hls::stream` objects, include the header file `hls_stream.h`. Streaming data objects are defined by specifying the type and variable name. In this example, a 16-bit signed fixed-point type is defined and used to create a stream variable called `my_wide_stream` as shown in listing 4.3.

```
#include "ap_int.h"
#include "hls_stream.h"
typedef ap_fixed<16,8> fp_data_type // 16-bit user-defined fixed-point
    datatype
hls::stream<fp_data_type> my_wide_stream; // A stream declaration
```

Listing 4.3: Streaming example

4.3.2 Naming streams

Optionally, streams can be given names that can be used in reporting. For instance, Vivado HLS automatically verifies that all elements from an input stream are read during simulation. This is often used in debugging scenarios where there is an issue with a stream not being read and there is leftover data that needs to be addressed. By providing a name for the stream, it can be easily identified and tracked to help identify the source of the issue as shown in listing 4.4 and 4.5.

```
#include "hls_stream.h"
// Unnamed stream
stream<fp_data_type> I1;
// Named stream
stream<fp_data_type> I2("input_stream2");
```

Listing 4.4: Example code for creating named and unnamed streams

```
Warning: Hls::stream 'hls::stream<fp_data_typer>.1' contains leftover data,
    which may result in RTL simulation hanging.
Warning: Hls::stream 'input_stream2' contains leftover data, which may result
     in RTL simulation hanging.
```

Listing 4.5: HLS warning for data stream

4.3.3 I/O for streams

When the streams are transferred into and out of the functions, they must be passed as a reference as shown in listing 4.6. In Vivado, it is possible to determine whether a variable should be stored in on-chip registers or off-chip memory. When a variable is passed by reference, Vivado generates a reference to the original variable and enables you to indicate whether that reference should be saved in on-chip registers or off-chip memory. In case of data being fetched from off-chip memory, Vivado will synthesize additional logic and bus interfaces to facilitate data transactions between the FPGA and off-chip memory.

```
void stream_function (
  hls::stream<fp_data_type> &yolo_stream_in,
  hls::stream<fp_data_type> &yolo_stream_out,
  uint16_t strm_len
)
{
  // Function body goes here
}
```

Listing 4.6: I/O stream example

4.3.4 Blocking reads and writes

The fundamental ways to interact with an `hls::stream<>` object involve performing blocking reads and writes through specific class methods. In the blocking read and write method, if an attempt is made to read from an empty stream FIFO or write to a full stream FIFO, these methods will cause execution to stall. Listing 4.7 gives an example of blocking read, and write. During C++ simulation warnings are generated when reading from an empty stream FIFO or writing to a full stream FIFO is made. During C/RTL co-simulation, this stall can be observed when the simulator continues to execute but no progress is made in the transactions being simulated. As a result, the simulation can become stuck or unresponsive, leading to delays or errors in the design.

```
// Usage of void read(T &rdata)
hls::stream<int> my_stream;
int dst_var;
my_stream.read(dst_var);

// Usage of void write(const T & wdata)
hls::stream<int> my_stream;
int src_var = 42;
my_stream.write(src_var);
```

Listing 4.7: Block read and write example

4.3.5 Non-Blocking reads and writes

Non-Blocking methods allow execution to continue even when attempting to read from an empty stream or write to a full stream. This is achieved by returning a status indicating whether the operation was successful or not, without blocking the execution of the program as shown in listing 4.8. Additional methods are available for checking the status of `hls::stream<>` objects before performing read and write operations. These methods check whether the stream is full before writing into the stream, and they also check whether the stream is empty before attempting to read from the stream. The methods are called the *Emptiness test* and *Fullness test* respectively as shown in line 5 and line 10 in listing 4.9. These methods can be used to determine if it is safe to perform a read or write operation thereby helping to avoid stalling the execution of the simulation.

```
// Usage of void write(const T & wdata)
hls::stream<int> my_stream;
int src_var = 42;
if (my_stream.write(src_var)) {
// Perform standard operations
} else {
 // Write did not occur
}
// Usage of void read(const T & wdata)
int dst_var;
if (my_stream.read(dst_var)) {
 // Perform standard operations
 ...
} else {
 // Read did not occur
 return;
}
```

Listing 4.8: Non-blocking read and write example

```
// Usage of bool full(void) (Non-Blocking writes)
hls::stream<int> my_stream;
int src_var = 42;
bool stream_full;
stream_full = my_stream.full();

// Usage of bool empty(void) (Non-Blocking reads)
int dst_var;
bool stream_empty;
stream_empty = my_stream.empty()

}
```

Listing 4.9: Non-blocking read and writes methods

4.4 Source code Example

This section discusses writing C++ code in Vivado using an example IP which is called in the design an "Accumulation and activation block". The IP is responsible for processing 32 input and output channels in one single execution but can execute 4 channels in parallel due to DMA constraints (64 bits) and fixed-point quantization. The detailed description of this block is given in chapter 8. This block is responsible for accumulating the result of the current convolution with the previously stored result of convolution in case of input channel folding. The concept of channel folding is explained in chapter 6. The code defines a top-level function *yolo_acc_top* with 2 streams as input: stream a takes the current result of convolution and stream b is responsible for streaming bias inside a local bias memory followed by

streaming the previous execution of 32 channels of convolution. Both streams are of type *yolo_quad_stream* which is a custom AXI interface for sending 64 bits streams. The creation of a custom AXI interface and how it gets mapped into RTL ports will be discussed in the next section. The function takes layer height *input_h* and layer width *input_w* as an input with an unsigned integer datatype with width 9. This is because the maximum input layer dimension comes from the first layer of yolov4-tiny which is 416×416. Input *fold_input_ch* takes into account the folding when 4 channels are processed in parallel. This is a set of 8 for processing 32 channels in one execution. Input *leaky* of type unsigned integer of width 1 takes into account the leaky activation function to be applied on 32 output channels after accumulation has been performed across all the input channels of the layer. Input *bias* adds the bias offset across the 32 output channels.

The IP block consists of 2 loops: loop 1 (line 15) is responsible for storing bias in local memory *kernel_bias_fp* which is an array of *MAX_KERNEL_NUM* which is set to 32, and loop 2 (line 30) is responsible for accumulation across input channels. The function *post_process* (line 81) is responsible for activation and bias addition after accumulation.

```
1  #include "yolo_acc.h"
2
3
4  void yolo_acc_top(yolo_quad_stream & inStream_a, yolo_quad_stream &
       inStream_b,
5    yolo_quad_stream & outStream,
6    ap_uint < 9 > input_h, ap_uint < 9 > input_w,
7    ap_uint < MAX_FOLD_CH_BIT > fold_input_ch,
8    ap_uint < 1 > leaky, ap_uint < 1 > bias_en) {
9
10   fp_weight_type kernel_bias_fp[MAX_KERNEL_NUM];
11
12   // loop 1 for storing bias
13   yolo_acc_top_label0: for (ap_uint < MAX_FOLD_CH_BIT > i = 0; i <
       fold_input_ch; i++) {
14     if (bias_en == 1) {
15       quad_fp_side_channel curr_input;
16       curr_input = inStream_b.read();
17       kernel_bias_fp[4 * i] = curr_input.data.sub_data_0;
18       kernel_bias_fp[4 * i + 1] = curr_input.data.sub_data_1;
19       kernel_bias_fp[4 * i + 2] = curr_input.data.sub_data_2;
20       kernel_bias_fp[4 * i + 3] = curr_input.data.sub_data_3;
21     }
22   }
23
24   // loop 2 for Accumulation and activation.
25   yolo_acc_top_label2: for (int row_idx = 0; row_idx < input_h; row_idx++) {
26     yolo_acc_top_label3: for (int col_idx = 0; col_idx < input_w; col_idx++)
       {
27       yolo_acc_top_label1: for (int input_ch_idx = 0; input_ch_idx <
       fold_input_ch; input_ch_idx++) {
28         quad_fp_side_channel curr_input_a, curr_input_b;
```

```
29          quad_fp_side_channel curr_output;
30
31          fp_data_type output_acc_0, output_acc_1, output_acc_2, output_acc_3;
32
33          curr_input_a = inStream_a.read(); // Current conv result
34          curr_input_b = inStream_b.read(); // previous iteration result
35
36          // Accumulation across 4 channels in parallel
37          output_acc_0 = curr_input_a.data.sub_data_0 + curr_input_b.data.
    sub_data_0;
38          output_acc_1 = curr_input_a.data.sub_data_1 + curr_input_b.data.
    sub_data_1;
39          output_acc_2 = curr_input_a.data.sub_data_2 + curr_input_b.data.
    sub_data_2;
40          output_acc_3 = curr_input_a.data.sub_data_3 + curr_input_b.data.
    sub_data_3;
41
42          // Activation and bias addition if bias_en = 1
43          curr_output.data.sub_data_0 = post_process_unit(output_acc_0,
    kernel_bias_fp[4 * input_ch_idx], bias_en, leaky);
44          curr_output.data.sub_data_1 = post_process_unit(output_acc_1,
    kernel_bias_fp[4 * input_ch_idx + 1], bias_en, leaky);
45          curr_output.data.sub_data_2 = post_process_unit(output_acc_2,
    kernel_bias_fp[4 * input_ch_idx + 2], bias_en, leaky);
46          curr_output.data.sub_data_3 = post_process_unit(output_acc_3,
    kernel_bias_fp[4 * input_ch_idx + 3], bias_en, leaky);
47
48          curr_output.keep = curr_input_a.keep;
49          curr_output.strb = curr_input_a.strb;
50          curr_output.user = curr_input_a.user;
51
52          //Check if all 32 channels are processed
53          if ((input_ch_idx == MAX_KERNEL_NUM / 4 - 1) &&
54            (col_idx == input_w - 1) &&
55            (row_idx == input_h - 1))
56            curr_output.last = 1;
57          else
58            curr_output.last = 0;
59
60          curr_output.id = curr_input_a.id;
61          curr_output.dest = curr_input_a.dest;
62
63          outStream.write(curr_output);
64
65        }
66      }
67    }
68
69 }
70
```

```
71 fp_data_type post_process_unit(fp_data_type data_in, fp_weight_type bias,
       ap_uint < 1 > bias_en, ap_uint < 1 > leaky) {
72   fp_data_type biased_output = 0, activated_output = 0;
73   if (bias_en) {
74     biased_output = data_in + bias;
75     if (leaky && biased_output < 0) {
76       activated_output = biased_output * (fp_data_type) .1;
77     } else {
78       activated_output = biased_output;
79     }
80
81     return activated_output;
82   } else {
83     return data_in;
84   }
85 }
```

Listing 4.10: Accumulation and activation IP

4.4.1 Pragmas and performance improvement

With custom datatypes and functional descriptions of the design, C++ descriptions can be made as shown in figure 4.10. To optimize the description to be converted into hardware, pragmas can be used. These are the additional directives that indicate to the compiler how the C++ description should be applied to the hardware. Vivado provides different pragmas to indicate different functionality and only relevant pragmas that are used for the implementation of the IP blocks are discussed. For instance, **#pragma HLS LOOP_TRIPCOUNT** is used by the Vivado HLS tool to calculate and provide the total latency of each loop, which represents the number of clock cycles that it takes to execute all iterations of that loop. The loop latency is dependent on the number of loop iterations, or tripcount. However, in certain cases, the HLS tool may not be able to compute the loop trip count when variables are either input variables or their values are calculated by dynamic operations. In such cases, it is possible to provide minimum and maximum values for those variables as additional arguments to the tool using directives tcl as shown in figure 4.2 (lines 16 to 19). The HLS tool can use these values as bounds to estimate the possible range of loop trip counts. As discussed earlier, *yolo_quad_stream* is a custom AXI interface used for streams a and b for sending 64 bits inputs. In order to create a custom AXI interface, the "hls_stream.h" line in the code is used with the hls::stream class, which is part of the High-Level Synthesis (HLS) library provided by Xilinx as shown in listing 4.11. The class hls::stream is utilized to create buffers that behave like first-in, first-out (FIFO) queues for data transmission between various hardware modules within an FPGA design. **#pragma HLS LOOP_INTERFACE** has been used to map the interface of the design to the RTL port. As can be seen from figure 4.2 (line 8-15), ports can be mapped as either AXI4-lite interface or AXI4-Stream interface depending upon the mode. The bundle option combines all the AXI4-Lite interfaces into interface ports in RTL code. In this example, *my_axi_interface* is a struct that represents a custom AXI

interface that is sending 64 bits of data. It has four template parameters: D for the data width in bits, U for the user width in bits, TI for the ID width in bits, and TD for the destination width in bits. The struct contains the following members:

- **data**: a custom data type that represents the data being transmitted over the interface.
- **keep**: an ap_uint that indicates which bytes of the data are valid.
- **strb**: an ap_uint that indicates which bytes of the data are being transmitted.
- **user**: an ap_uint that carries user-defined information.
- **last**: a signal that indicates whether the current transaction is the last in a burst.
- **id**: an ap_uint that identifies the transaction.
- **dest**: an ap_uint that identifies the destination of the transaction.

Line 21 creatures custom AXI interface with name *yolo_quad_stream* of type *my_axi_interface*

```
1  #include "ap_int.h"
2  #include "hls_stream.h"
3
4  typedef struct my_data_type {
5  ap_fixed<16,8> a;
6  ap_fixed<16,8> b;
7  ap_fixed<16,8> c;
8  ap_fixed<16,8> d;
9  } my_data_type;
10
11 template<int D, int U, int TI, int TD>
12 struct my_axi_interface {
13 my_data_type data;
14 ap_uint<(D+7)/8> keep;
15 ap_uint<(D+7)/8> strb;
16 ap_uint<U> user;
17 ap_uint<1> last;
18 ap_uint<TI> id;
19 ap_uint<TD> dest;
20 };
21 typedef hls::stream<my_axi_interface<64, 4, 5, 6>> yolo_quad_stream;
```

Listing 4.11: Custom AXI interface definition

```
8  set_directive_interface -mode s_axilite -bundle CTRL_BUS "yolo_acc_top" bias_en
9  set_directive_interface -mode s_axilite -bundle CTRL_BUS "yolo_acc_top" leaky
10 set_directive_interface -mode s_axilite -bundle CTRL_BUS "yolo_acc_top" input_w
11 set_directive_interface -mode s_axilite -bundle CTRL_BUS "yolo_acc_top" fold_input_ch
12 set_directive_interface -mode s_axilite -bundle CTRL_BUS "yolo_acc_top" input_h
13 set_directive_interface -mode axis -register -register_mode both "yolo_acc_top" inStream_a
14 set_directive_interface -mode axis -register -register_mode both "yolo_acc_top" inStream_b
15 set_directive_interface -mode axis -register -register_mode both "yolo_acc_top" outStream
16 set_directive_loop_tripcount -min 8 -max 8 "yolo_acc_top/yolo_acc_top_label0"
17 set_directive_loop_tripcount -min 208 -max 208 "yolo_acc_top/yolo_acc_top_label2"
18 set_directive_loop_tripcount -min 208 -max 208 "yolo_acc_top/yolo_acc_top_label3"
19 set_directive_loop_tripcount -min 8 -max 8 "yolo_acc_top/yolo_acc_top_label1"
```

Figure 4.2: Setting loop bounds using HLS TRIP_COUNT

4.4.1.1 Performance with no pipelining

For a tunable design, it is important to check which design parameter can be tuned to achieve lower latency with as low resource utilization as possible. Vivado provides different solutions pertaining to incremental improvement in the design which makes design choices easier. These incremental performance improvements come with pipelining and memory partitioning of the design. The design example in 4.10 is synthesized at 100 MHz clock frequency with no optimization first to see worst-case latency as shown in figure 4.3. This is named as *solution 1*. The worst-case latency of the design without pipelining is 28.32 ms and is equal to the loop latency of the individual loops under the Loop report shown in figure 4.3. This solution has not been pipelined as shown in the report

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2855858	2855858	28.559 ms	28.559 ms	2855858	2855858	none

Detail

Instance

		Latency (cycles)		Latency (absolute)		Interval (cycles)		
Instance	Module	min	max	min	max	min	max	Type
grp_post_process_unit_fu_743	post_process_unit	2	2	20.000 ns	20.000 ns	2	2	none
grp_post_process_unit_fu_751	post_process_unit	2	2	20.000 ns	20.000 ns	2	2	none
grp_post_process_unit_fu_759	post_process_unit	2	2	20.000 ns	20.000 ns	2	2	none
grp_post_process_unit_fu_767	post_process_unit	2	2	20.000 ns	20.000 ns	2	2	none

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- yolo_acc_top_label0	16	16	2	-	-	8	no
- yolo_acc_top_label2	2855840	2855840	13730	-	-	208	no
+ yolo_acc_top_label3	13728	13728	66	-	-	208	no
++ yolo_acc_top_label1	64	64	8	-	-	8	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	643	-
FIFO	-	-	-	-	-
Instance	-	4	264	980	-
Memory	1	-	0	0	0
Multiplexer	-	-	-	443	-
Register	-	-	351	-	-
Total	1	4	615	2066	0
Available	280	220	106400	53200	0
Utilization (%)	~0	1	~0	3	0

Figure 4.3: Latency and resource estimation

4.4.1.2 Performance and resource utilization with pipelining

Figure 4.4 and 4.5 show the analysis view of the design which shows the latency interval of each loop. It is a useful tool to identify the bottleneck operations in the design as this tool specifies the operations and how they are scheduled. Vertical axis with C_1,C_2 up to C_{10} shows the clock cycles where certain operations are active. As the local memory is implemented using 2 port BRAM, it takes 2 clock cycles for loop 1 to write to $kernel\_bias\_fp$ as shown in figure 4.4. Pipelining optimization is applied to both Loop 1 and the inner loop of Loop 2 to enhance performance. This optimization can be achieved by creating a new solution in Vivado and specifying the pipelining optimization directive with a target initiation interval of one clock cycle. The impact of this optimization is depicted in Figure 4.4, where the Gantt chart illustrates that after pipelining both loops, the initiation interval of each loop becomes 2 clock cycles. Due to the implementation of a 2-port BRAM for local memory, which has not been partitioned, memory writes occur for only the first 2 bias

memory locations concurrently, rather than 4 different locations per iteration. Consequently, it takes 2 clock cycles to complete the write operations for all 4 memory locations. Similarly, in Loop 2 as shown in figure 4.5, limited memory ports prevent concurrent biased memory read operations for the last 2 channels within a single control step, thereby violating the target initiation interval of 1 clock cycle. The introduction of pipelining has significantly reduced the latency of the design from 28.2 ms to 6.992 ms but does not achieve an initiation interval of 1 clock cycle. However, further parallelization can be achieved by partitioning the biased memory, enabling all reads to be performed in a single control step for Loop 2 (lines 52,53,54,55) and facilitating the concurrent memory write at 4 different biased memory locations within a single control step for Loop 1 (lines 21,22,23,24).

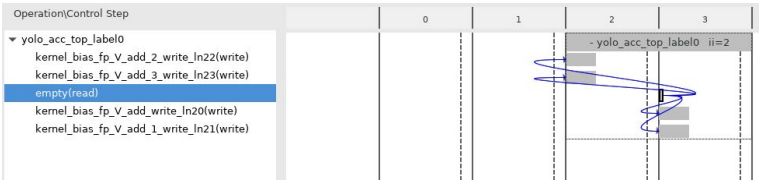


Figure 4.4: Loop 1 improved initiation interval with pipelining: solution 2

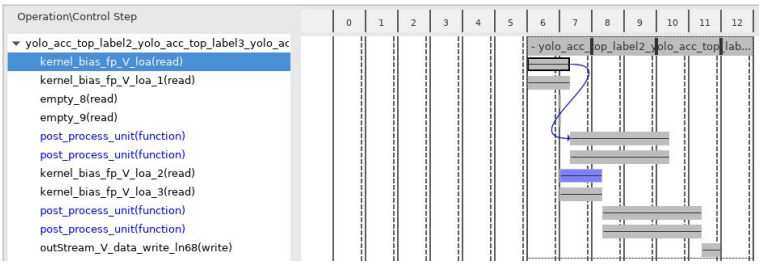


Figure 4.5: Loop 2 improved initiation interval with pipelining: solution 2

4.4.1.3 Performance and resource utilization with pipelining and bias memory partiton

After identifying the bottleneck read and write operation that resulted in a 2-clock cycle initiation interval for the loop, the next step is to address this issue by partitioning the memory. Figure 4.6, and 4.6 the effect of the memory partitioning process, where the memory is divided cyclically with a partitioning factor of 2. As a result, two separate arrays of size 16 are created, allowing two memory locations to be fetched simultaneously from each array. This enables concurrent reading and writing of data during the same control step, effectively reducing the initiation interval for both loops and achieving the target initiation interval of 1 clock cycle. This is depicted in figure 4.6 and 4.7

4.4.2 Resource utilization comparision

Vivado provides resource utilization and latency estimation based on the optimization applied in the C++ description. The resource utilization report depicted in Figure [X] provides insights into the utilization of resources for three different solutions. The number of Block

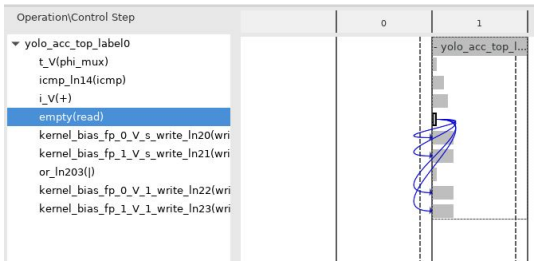


Figure 4.6: Loop 1 improved initiation interval with pipelining and memory partition: solution 3

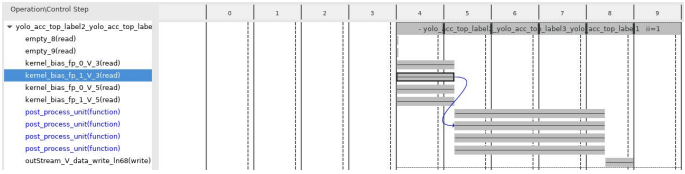


Figure 4.7: Loop 2 improved initiation interval with pipelining and memory partiton: solution 3

RAMs (BRAMs) inferred depends on the memory partitioning. Since each BRAM can store 1k words of size 16 bits, it is sufficient to accommodate 32 biases for the 32 output channels. In the case of "solution 1" and "solution 2" where no partitioning is applied, 1 BRAM block is inferred. However, for "solution 3" which involves memory partitioning by a factor of 2, 2 BRAM is needed. Regarding the inferred DSP blocks, "solution 3" stands out as it processes all four channels in parallel using four different post-processing functional unit blocks. This results in the inference of four DSP blocks. Additionally, one extra DSP block is attributed to the control logic within the code. There exists a tradeoff between resource utilization and latency. While "solution 3" achieves the minimum latency, it exhibits slightly higher utilization of Look-Up Tables (LUTs) and Flip-Flops (FFs) compared to the other solutions. The memory partitioning factor is a tunable parameter that controls the latency as well as the resource utilization. This is discussed in chapter 6.

Performance Estimates

Timing

Clock		solution3	solution1	solution2
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns
	Estimated	8.632 ns	7.667 ns	8.632 ns

Latency

		solution3	solution1	solution2
Latency (cycles)	min	346129	2855858	692249
	max	346129	2855858	692249
Latency (absolute)	min	3.461 ms	28.559 ms	6.922 ms
	max	3.461 ms	28.559 ms	6.922 ms
Interval (cycles)	min	346129	2855858	692249
	max	346129	2855858	692249

Utilization Estimates

	solution3	solution1	solution2
BRAM_18K	2	1	1
DSP48E	5	4	3
FF	1381	615	1231
LUT	2234	2066	1930
URAM	0	0	0

Figure 4.8: Resource utilization comparision

Chapter 5

Software Implementation of YOLOv4-tiny on Zedboard's PS and Host CPU

This chapter provides an overview of the software implementation of the Yolov4-tiny algorithm using the darknet framework and discusses the main computations in each layer present in the Yolov4-tiny model. Then a detailed analysis of the performance of the YOLOv4-tiny algorithm on various hardware platforms, including desktop CPUs and ZedBoard's arm processor is discussed by profiling the code on these platforms. Section 5.1 overviews the main computations in each layer. Then, the profiling results of running the model on the host CPU are discussed in section 5.2. Then, section 5.3 discusses executing the same model inside the processing system (Processing System (PS)) of the Zedboard to identify computationally intensive functions. This identification helps to create a hardware-software codesign wherein a computationally intensive function is executed inside the programmable logic (Programmable Logic (PL)) section of the hardware and the rest of the less time taking tasks are handled by the PS only. The host CPU and PS of the Zedboard are selected as a benchmark for performance comparison. Finally, section 5.4 concludes with a discussion of the YOLOv4-tiny bare-metal fixed-point model, which is an alternative implementation with reduced precision that becomes the baseline model for the hardware-accelerated model.

5.1 Software Implementation

The project is aimed at the ZedBoard, which has a dual-core ARM CortexA9-based processing system (PS) that operates at a maximum frequency of 667 MHz. To simplify the development of the hardware accelerator, only one core is used to run a bare-metal software implementation. The rest of this section discusses about the floating point model implementation details and its performance when running on a host CPU with core-i5 specification running at 1.2 GHz. The code is ported to the ARM A9 cortex processor of Zedboard and profiling results are obtained to analyze the most computationally intensive layers. Finally, A fixed point model based is discussed whose bit-width is decided based on the loss function.

5.1.1 Yolov4-tiny Baremetal Floating-point model

The YOLOv4-tiny object detection model has been implemented as a bare-metal floating-point model which runs directly on hardware without the need for an operating system or software layer. This implementation utilizes floating-point operations to represent numerical values, enabling higher accuracy calculations compared to using integer operations. The Darknet framework has been used to implement the bare-metal version of the YOLOv4-tiny model. Darknet is an open-source neural network framework written in C and CUDA that is used for a variety of computer vision tasks, including object detection and classification. It is lightweight and fast, making it a good choice for implementing neural network models on resource-constrained devices. The darknet framework has been modified to not include the camera interface and file operations to suit the bare-metal application.

5.1.1.1 Implementation

Convolutional Layer
The forward propagation for a convolutional layer, as shown in Fig 5.1 involves the following steps:

- **Convolution operation:** To perform the convolution operation in a convolutional layer, the input tensor is convolved with learnable filters. Darknet implements this operation using the im2col function, which converts the input tensor into a matrix and performs matrix multiplication with the filter weights. The resulting output is then reshaped back into a tensor. The `gemm_nn()` (Line 13) function call is used to execute this operation.
- **Bias Addition:** After the convolution operation, biases are added to each filter in a process known as bias addition. This is carried out using the `add_bias()` (Line 31)function call.
- **Activation Function:** The activation function introduces non-linearity in the model, the output of the convolutional layer is passed through an activation function. The modified Darknet supports several activation functions, including linear, leaky-ReLU, and logistic The `activate_array()` (Line 31) function call is used to execute this step.

This code in figure 5.1 represents the forward propagation of a convolutional layer in a neural network. The function takes a convolutional layer and a network as input and fills the output buffer with the result of the convolution. The code initializes variables to define the shape and size of the convolutional layer and then performs matrix multiplication using the layer weights and input data. It initializes the output to zero and calculates the number of kernels, kernel size, and a number of outputs. It then loads the layer input to the workspace using `im2col_cpu()`, which converts the input into a matrix so that matrix multiplication can be performed. The matrix multiplication is performed using the `gemm()` function, which computes the product of matrices. Finally, batch normalization is performed if specified, biases are

```
void forward_convolutional_layer(convolutional_layer l, network net)
{
    fill_cpu(l.outputs*l.batch, 0, l.output, 1);    //clear outpu
    int m = l.n/l.groups;   //number of kernels
    int k = l.size*l.size*l.c/l.groups; //kernel size * channel
    int n = l.out_w*l.out_h;    //number of outputs
    fprintf(stderr,"(M:%d,K:%d,N:%d)\n",m,k,n);
            float *a = l.weights;
            float *b = net.workspace;
            float *c = l.output;
            float *im =  net.input;
            if (l.size == 1) {
                b = im;
            } else {
                im2col_cpu(im, l.c/l.groups, l.h, l.w, l.size, l.stride, l.pad, b); //load layer input to the workspace
            }
            float *b_t = calloc(n*k,sizeof(float));
            for(int bi=0;bi<k;bi++)
            {
                for(int bj=0;bj<n;bj++)
                {
                    b_t[bj*k+bi] = b[bi*n+bj];
                }
            }
            gemm(0,1,m,n,k,1,a,k,b_t,k,1,c,n); // General Matrix Multiplication
            free(b_t);

    if(l.batch_normalize){
        forward_batchnorm_layer(l, net);
    } else {
        add_bias(l.output, l.biases, l.batch, l.n, l.out_h*l.out_w);
    }
    activate_array(l.output, l.outputs*l.batch, l.activation);

}
```

Figure 5.1: Code Snippet of Forwarding of Convolutional Layer

added, and the activation function is applied to the output.

Image to column function (Im2col)! ((Im2col)!):

The `im2col()` function in Darknet is used to convert the input tensor (which has dimensions `(batch, channels, height, width)` into a matrix form with dimensions `(channels * kernel_height * kernel_width, output_height * output_width)`. This matrix form can be efficiently multiplied with the filter weights using the `gemm()` operation. The `im2col()` function in Darknet applies a sliding window of size `(kernel_height, kernel_width)` over the input tensor and copies the values within the window into a column of the output matrix. This process is repeated for each channel in the input tensor, resulting in a matrix where each column represents a sliding window of the input tensor. The `im2col()` function in Darknet also performs a padding operation on the input tensor before sliding the window to optimize memory usage. This ensures that the output matrix has the same dimensions for all input tensors, regardless of their original spatial dimensions. The padded values are initialized to 0 and are not used in the subsequent convolution operation. Overall, the `im2col()` operation in Darknet improves the efficiency of the convolutional layer by transforming the input tensor into a matrix form that can be multiplied efficiently with the filter weights using the Gemm operation, while still maintaining the same number of parameters.

General Matrix to Matrix Multiplication gemm():

GEMM is an acronym for "General Matrix to Matrix Multiplication." It is a mathematical operation that involves multiplying two matrices to create a third matrix [37]. In deep

learning, GEMM is commonly utilized to execute the convolution operation in convolutional neural networks (CNNs), which is a vital component of numerous state-of-the-art computer vision models. To achieve high computational efficiency on CPUs, the GEMM operation is typically executed with highly optimized numerical libraries such as BLAS (Basic Linear Algebra Subprograms) or openBLAS. Additionally, `gemm()` function in Darknet is parallelized to take advantage of multi-core CPUs, further improving its speed efficiency.

Maxpool Layer:

```
void forward_maxpool_layer(const maxpool_layer l, network net)
{
    int b,i,j,k,m,n;
    int w_offset = -l.pad/2;
    int h_offset = -l.pad/2;

    int h = l.out_h;
    int w = l.out_w;
    int c = l.c;

    for(b = 0; b < l.batch; ++b){
        for(k = 0; k < c; ++k){
            for(i = 0; i < h; ++i){
                for(j = 0; j < w; ++j){
                    int out_index = j + w*(i + h*(k + c*b));
                    float max = -FLT_MAX;
                    int max_i = -1;
                    for(n = 0; n < l.size; ++n){
                        for(m = 0; m < l.size; ++m){
                            int cur_h = h_offset + i*l.stride + n;
                            int cur_w = w_offset + j*l.stride + m;
                            int index = cur_w + l.w*(cur_h + l.h*(k + b*l.c));
                            int valid = (cur_h >= 0 && cur_h < l.h &&
                                         cur_w >= 0 && cur_w < l.w);
                            float val = (valid != 0) ? net.input[index] : -FLT_MAX;
                            max_i = (val > max) ? index : max_i;
                            max   = (val > max) ? val   : max;
                        }
                    }
                    l.output[out_index] = max;
                    l.indexes[out_index] = max_i;
                }
            }
        }
    }
}
```

Figure 5.2: Code Snippet of Forward Propagation for Maxpool Layer

This code represents the implementation of the forward pass for a max pooling layer in a convolutional neural network. The max pooling operation is applied to each feature map separately. The function takes two arguments:

1. `"l"` is an object of type maxpool_layer, which contains the configuration parameters of the layer such as the size of the pooling window, the stride, the padding, etc.
2. `"net"` is an object of the type network, which contains the input data for the layer and will be updated with the output of the layer.

The function loops over the input data in batches, feature maps, rows, and columns, and for each window, it finds the maximum value. It also keeps track of the index of the maximum value in the input array for each window. The output of the layer is stored in the output field of l, while the indices of the maximum values are stored in the indexes field of l. Note that the code uses the `"FLT_MAX` constant from the `float.h` library to initialize the max variable with the smallest possible float value.

Upsample Layer:

This code implements the forward pass for an upsample layer in a convolutional neural network. The function takes two arguments:

1. "l" is an object of type layer, which contains the configuration parameters of the layer such as the scale factor and the stride.
2. "net" is an object of type network, which contains the input data for the layer and will be updated with the output of the layer.

The function first fills the output array with zeros using the `fill_cpu` function. Then it

```
void upsample_cpu(float *in, int w, int h, int c, int batch, int stride, int forward,  float scale, float *out)
{
    int i, j, k, b;
    for(b = 0; b < batch; ++b){
        for(k = 0; k < c; ++k){
            for(j = 0; j < h*stride; ++j){
                for(i = 0; i < w*stride; ++i){
                    int in_index = b*w*h*c + k*w*h + (j/stride)*w + i/stride;
                    int out_index = b*w*h*c*stride*stride + k*w*h*stride*stride + j*w*stride + i;
                    if(forward) out[out_index] = scale*in[in_index];
                    else in[in_index] += scale*out[out_index];
                }
            }
        }
    }
}

void forward_upsample_layer(const layer l, network net)
{
    fill_cpu(l.outputs*l.batch, 0, l.output, 1);
    upsample_cpu(net.input, l.w, l.h, l.c, l.batch, l.stride, 1, l.scale,l.output);

}
```

Figure 5.3: Code Snippet of Forward Propagation for Upsample Layer

calls the `upsample_cpu` function to perform the actual upsampling operation. The `upsample_cpu` function loops over the input data in batches, feature maps, rows, and columns, and for each input pixel, it computes the corresponding output pixel using the specified stride and scale factor. If the forward flag is set to true, it stores the result in the out array. Otherwise, it accumulates the result in the array.

Route Layer:

This code represents the implementation of the forward pass for a route layer in a convolutional neural network. The layer concatenates the outputs of multiple previous layers. The function takes two arguments:

1. "l" is an object of type route_layer, which contains the configuration parameters of the layer such as the input layers and their sizes, the number of input layers, the output size, etc.
2. "net' is an object of type network, which contains the input data for the layer and will be updated with the output of the layer.

```c
void forward_route_layer(const route_layer l, network net)
{
    int i, j;
    int offset = 0;
    for(i = 0; i < l.n; ++i){
        int index = l.input_layers[i];
        float *input = net.layers[index].output;
        int input_size = l.input_sizes[i];
        int part_input_size = input_size / l.groups;
        for(j = 0; j < l.batch; ++j){
            copy_cpu(part_input_size, input + j*input_size + part_input_size*l.group_id, 1, l.output + offset + j*l.outputs, 1);
        }
        offset += part_input_size;
    }
}
```

Figure 5.4: Code Snippet of Forward Propagation for Route Layer

The function loops over the input layers specified in the input_layers field of l and copies their outputs to the output field of l. The output is concatenated along the channel dimension, which is split into multiple groups (specified in the group's field of l). Each input layer's output is split into the same number of groups, and only one group is copied at a time, determined by the current group_id of l. The offset variable is used to keep track of the current position in the output array.

Yolo Layer:

This code represents the implementation of the forward pass for a YOLO layer in a convolutional neural network. The function takes two arguments:

1. l is an object of type yolo_layer, which contains the configuration parameters of the layer such as the number of anchors, the number of classes, etc.
2. net is an object of type network, which contains the input data for the layer and will be updated with the output of the layer.

The function first copies the input data from the net to the output field of l. Then it loops over each batch and each anchor and applies the logistic activation function to certain elements of the output based on their index. Specifically, for each anchor, it applies logistic activation to the confidence scores (located at index 0) and the class probabilities (located at index 4) for each cell in the grid.

```c
void forward_yolo_layer(const layer l, network net)
{
    int b,n;
    memcpy(l.output, net.input, l.outputs*l.batch*sizeof(float));
    for (b = 0; b < l.batch; ++b){
        for(n = 0; n < l.n; ++n){
            int index = entry_index(l, b, n*l.w*l.h, 0);
            activate_array(l.output + index, 2*l.w*l.h, LOGISTIC);
            index = entry_index(l, b, n*l.w*l.h, 4);
            activate_array(l.output + index, (1+l.classes)*l.w*l.h, LOGISTIC);
        }
    }
}
```

Figure 5.5: Code Snippet of Forward Propagation for Route Layer

5.2 Profiling results for yolov4-tiny on Host CPU

% time	Cum s	self s	self calls	self s/call	total s/call	Name
88.08	7.75	7.75	21	0.37	0.37	gemm_nt
4.21	8.12	0.37	21	0.02	0.41	forward_convolutional_layer
1.82	8.28	0.16	14	0.01	0.02	im2col_cpu
1.36	8.40	0.12	60	0.00	0.00	fill_cpu
1.25	8.51	0.11	29592576	0.00	0.00	im2col_get_pixel
0.68	8.57	0.06	37	0.00	0.00	copy_cpu
0.68	8.63	0.06	19	0.00	0.00	normalize_cpu
0.45	8.67	0.04	19	0.00	0.00	scale_bias
0.34	8.70	0.03	45	0.00	0.00	activate_array
0.34	8.73	0.03	21	0.00	0.00	add_bias
0.34	8.76	0.03	3	0.01	0.01	forward_maxpool_layer
0.23	8.78	0.02	6758141	0.00	0.00	activate
0.11	8.79	0.01	215475	0.00	0.00	linear_activate
0.11	8.80	0.01	21	0.00	0.37	gemm_cpu

Table 5.1: Flat Profile

Table 5.1 provides the profiling results of the yolov4-tiny model running on the host CPU (Corei5) with 4 cores running at a frequency of 1.2 GHz. The figure shows only those function that takes more than 10 ms of time to complete. The *time* column shows the percentage of the total running time of the program used by this function. The total Execution time of the model is given by the second column *Cumulative_secs*, which is the running sum of the number of seconds accounted for by this function and those listed above it. The third column, *self seconds* shows the several seconds accounted for by this function alone. This is the major sort for this listing. The fourth column depicts the number of times the function was called in an inference cycle. Column Fifth and Sixth depict the self-time per call and total time per call which denotes the average number of milliseconds spent in this function per call but in the case of the former the time spent is w.r.t to the function which is called and the latter is w.r.t to itself and it's descendant functions.

After analyzing the profiling results as presented in table 5.1, the most computationally intensive layer is the convolutional layer as the cumulative sum of all the functions inside the $forward\_convolutional\_layer$ comes out to be 99.67%. The generic matrix multiplication, shown in grey, takes about 88.08% of the execution times. Except for the $forward\_maxpool\_layer$ shown in red, all other functions called are part of the top function called $forward\_convolutional\_layer$. So the total time taken by the yolov4-tiny model on the host CPU is 8.8 secs.

5.3 Vitis Vivado setup and profiling results of yolov4-tiny model on ZedBoard PS

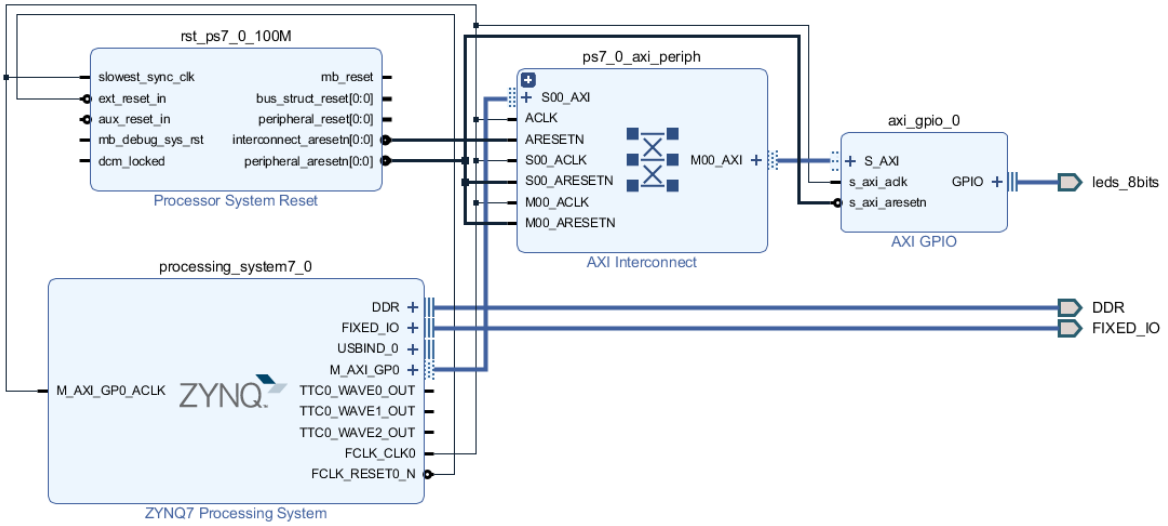


Figure 5.6: Vivado setup for Yolov4-tiny Floating point model

The Zynq System on Chip (SoC) is a versatile and powerful platform that combines multiple components, including Arm Cortex-A9 cores, pre-designed intellectual property (IP) modules, and programmable logic (PL). This integration enables the creation of flexible and customizable designs, where the processing system (PS) delivers the computing power and interface to external devices, while the PL can be programmed to provide extra hardware acceleration or tailor-made functionality. The Zynq device can be utilized in two distinct ways:

- The Zynq SoC PS can be used independently, without attaching any additional fabric IP.
- IP cores can be instantiated in fabric and combined with the Zynq PS as a PS+PL configuration.

Figure 5.6 shows the setup where the yolov4-tiny setup is in PS-only mode. The idea for this setup is to find out the time required by the various layers in the yolov4-tiny and then do the most computationally intensive task on the PL part of the FPGA and others inside the PS only. This Hardware-Software Codesign setup is discussed in the next section. The current setup with PS-only mode consists of the following blocks:

- **Zynq Processing System (PS)**: The PS block is a crucial component of the Zynq device and serves as both the processing power and interface to the outside world. It is made up of multiple subsystems, including the CPU cores, memory controller, and various peripherals like UART, Ethernet, USB, and SDIO. The PS block can be customized and configured to meet the particular needs of the application it is being

used for. In this bare-metal application, the peripherical used for displaying the output of the model is through UART. The output of the UART is connected to a serial terminal called "Tera Term" which serially prints the output of the UART output to the console.
please note that Revision E Zedboard was used for implementing the testing of the simulation model. It's also important to note that the frequency of the PS is set to 667 Mhz and the Memory part number for this board was MT41K256M16RE-125. Both these configurations are necessary for the setup to run.

- **rst_ps7_0_100M block**: The rst_ps7_0_100M block is used to reset the processing system and ensure that it starts in a known state. This block generates a reset signal that is used to initialize the PS block and its peripherals.

- **ps7_0_axi_peripheral block**: The ps7_0_axi_peripheral block offers an AXI interface that allows for the connection of custom IP cores to the PS. This interface adheres to the AXI standard, which is commonly used for communication between IP cores and the PS. It enables high-bandwidth, low-latency communication and facilitates the integration of custom IP cores into the Zynq device.

- **axi_gpio_0 block**: The axi_gpio_0 block is responsible for providing a set of configurable input/output (GPIO) pins that can be utilized to interface with external devices or sensors. It enables the Zynq device to be connected to a wide range of external components like switches, buttons, LEDs, or sensors, through its 32 programmable input/output pins that can be configured as either input or output.

The Yolov4-tiny Simulation model profiling results are displayed in Figure 5.7. The most computationally intensive functions in the model are the General Matrix Multiplication `gemm_nt()` when running the model in the host CPU. By profiling the application which is running in the PS section at a frequency of 667 Mhz, bottleneck layers can be found. The TCF profiler in Vitis helps in profiling the software code written for PS and it provides two performance metrics: "% inclusive" and "% exclusive" which measure the execution time spent within a function (inclusive) and within a function, not including the time spent in its sub-functions (exclusive), respectively. These metrics can help to identify performance bottlenecks and optimization opportunities in the application. The top function is $Forward\_convolutional\_layer$ which calls $gemm\_nt()$, taking about 93% of the time, while the rest is taken by other functions such as $im2col\_cpu$. The 5.8 shows the sub-function breakdown of $Forward\_convolutional\_layer$. The total time taken by all the sub-functions is approximately 98.3% of the total time. Table 5.2 shows the detailed execution time analysis of each layer.

TCF Profiler

Profiler running. 19060 samples

Address	% Exclusive	% Inclusive	Function	File	Line
0010cf00	,000	100,	_start	xil-crt0.S	59
00102560	,000	100,	main	helloworld.c	57
0010379c	,000	98,7	network_predict	network.c	26
00103910	,000	98,7	forward_network	network.c	54
00101e68	1,21	98,1	forward_convolutional_layer	convolutional_layer.c	131
001021f0	,000	93,0	gemm	gemm.c	8
001024f0	,000	93,0	gemm_cpu	gemm.c	87
001023c0	93,0	93,0	gemm_nt	gemm.c	33
0010d52c	,000	1,97	_calloc_r		
0010e0f6	1,97	1,97	memset		
00102860	,965	1,83	im2col_cpu	im2col.c	20
00109330	,000	1,26	parse_network_cfg	parser.c	1334
00100760	,000	1,04	forward_batchnorm_layer	batchnorm_layer.c	4
00104260	,000	,966	parse_convolutional	parser.c	59
0010197c	,000	,966	make_convolutional_layer	convolutional_layer.c	39
0010279c	,860	,860	im2col_get_pixel	im2col.c	6
00100978	,456	,619	normalize_cpu	blas.c	18
0010c160	,136	,289	sqrt		
00100870	,247	,247	fill_cpu	blas.c	6
001008e8	,231	,231	copy_cpu	blas.c	12
00107f50	,000	,226	parse_layer_28	parser.c	1017
001006d0	,089	,226	activate_array	activations.c	39
00103360	,205	,205	forward_maxpool_layer	maxpool_layer.c	36
00101854	,168	,168	add_bias	convolutional_layer.c	26
00104814	,000	,152	parse_layer_0	parser.c	145
001016c8	,147	,147	scale_bias	convolutional_layer.c	10
00100650	,100	,136	activate	activations.c	4
001044b8	,000	,131	parse_route	parser.c	97
00109f98	,000	,131	make_route_layer	route_layer.c	9
00106b94	,000	,115	parse_layer_18	parser.c	703
00105c08	,000	,110	parse_layer_10	parser.c	460
00105048	,000	,094	parse_layer_4	parser.c	277
0010a13c	,000	,094	forward_route_layer	route_layer.c	39
00107b20	,000	,073	parse_layer_26	parser.c	947
00104c44	,000	,052	parse_layer_2	parser.c	215
0010b1b0	,000	,047	forward_yolo_layer	yolo_layer.c	103
0010f9a4	,000	,042	_write_r		
0010cd7c	,021	,037	XUartPs_SendByte	xuartps_hw.c	57
001077ac	,000	,031	parse_layer_24	parser.c	894
00104a2c	,000	,026	parse_layer_1	parser.c	180
0010567c	,000	,026	parse_layer_7	parser.c	373
00105894	,000	,026	parse_layer_8	parser.c	407
001005b0	,026	,026	leaky_activate	activations.h	10
0010c470	,026	,026	__ieee754_sqrt		
00105478	,000	,021	parse_layer_6	parser.c	346
00108cc4	,000	,021	parse_layer_35	parser.c	1229
00112638	,000	,021	write	write.c	30
00106608	,000	,016	parse_layer_15	parser.c	617

Figure 5.7: Profiling results

Caller Functions

Address	% Exclusive	Function	File	Line
00103910	98,3	forward_network	network.c	54

Callee Functions

Address	% Exclusive	Function	File	Line
001021f0	92,6	gemm	gemm.c	8
00102860	2,72	im2col_cpu	im2col.c	20
00100760	,920	forward_batchnorm_layer	batchnorm_layer.c	4
0010d52c	,825	_calloc_r		0
001006d0	,280	activate_array	activations.c	39
00100870	,079	fill_cpu	blas.c	6
00101854	,005	add_bias	convolutional_layer.c	26

Figure 5.8: Bottleneck layer functions

No	Layer	Simulation Time (Sec)	% Execution time
1	Convolutional Layer 1	2.987	1.540
2	Convolutional Layer 2	11.660	6.030
3	Convolutional Layer 3	22.936	11.500
4	Route Layer 1	0.012	0.006
5	Convolutional Layer 4	6.225	3.210
6	Convolutional Layer 5	6.225	3.210
7	Route Layer 2	0.024	0.012
8	Convolutional Layer 6	2.773	1.430
9	Route Layer 3	0.048	0.020
10	Maxpool Layer 1	0.193	0.090
11	Convolutional Layer 7	21.932	11.350
12	Route Layer 4	0.006	0.003
13	Convolutional Layer 8	5.686	0.030
14	Convolutional Layer 9	5.686	0.030
15	Route Layer 5	0.012	0.006
16	Convolutional Layer 10	2.548	1.310
17	Route Layer 6	0.024	0.012
18	Maxpool Layer 2	0.097	0.040
19	Convolutional Layer 11	21.392	11.070
20	Route Layer 7	0.003	0.002
21	Convolutional Layer 12	5.446	2.815
22	Convolutional Layer 13	5.446	2.815
23	Route Layer 8	0.006	0.003
24	Convolutional Layer 14	2.427	1.250
25	Route Layer 9	0.012	0.006
26	Maxpool Layer 3	0.049	0.002
27	Convolutional Layer 15	21.171	10.950
28	Convolutional Layer 16	1.188	0.610
29	Convolutional Layer 17	10.600	5.480
30	Convolutional Layer 18	1.171	0.880
31	Yolo Layer 1	0.024	0.011
32	Route Layer 10	0.002	0.007
33	Convolutional Layer 19	0.304	0.157
34	Upsample Layer 1	0.019	0.009
35	Route Layer 11	0.009	0.004
36	Convolutional Layer 20	32.007	16.560
37	Convolutional Layer 21	2.372	1.220
39	Yolo Layer 2	0.094	0.004
Total Time		193.188 Secs	

Table 5.2: Profiling results YOLOv4-tiny on ARM cortex A9

From the table, it can be concluded that out of 38 Layers, 21 convolutional layers are computationally more intensive and takes 98.6% of the execution times as compared to other layers. The total execution time of the model running in Arm A-9 cortex is 193.38 secs and is image-independent.

5.4 Yolov4-tiny Fixed-point Model

The data and weights in a neural network, as discussed before, are typically stored as 32-bit floating point numbers, which can require significant computational power and memory storage. To address this challenge, a common approach is to use fixed-point representation instead, as long as the accuracy of the system is not significantly affected. Using the fixed-point representation of data and weights in a network can improve performance and efficiency in multiple ways. Firstly, fixed-point numbers require less memory storage and computation capability compared to floating-point numbers. This leads to more efficient use of available hardware resources, allowing for faster processing times and reduced power consumption. Secondly, fixed-point arithmetic is generally faster and more efficient to compute than floating-point arithmetic, as it involves simpler operations that can be executed more quickly. This leads to improved performance and reduced latency in processing the network.

The Xilinx `ap_fixed.h` library is utilized to efficiently implement fixed-point arithmetic in HLS. To make use of this library, four decisions must be made: the total number of bits, the number of bits for integer parts, quantization mode, and overflow mode. When transferring data via the 64-bit AXI4-stream protocol, using powers of 2 for total bit-width is preferred to simplify the encoding and decoding process. So possible bit widths are 4,8,16 bits. To determine the appropriate bit-width, experiments were conducted on the COCO2017-val5k dataset which consists of 5,000 images. The data distribution of all convolutional layers is summarized in Appendix A.42, which shows that the absolute values of all inputs and outputs are less than 128. Therefore, allocating 8 bits to the signed integer part is sufficient to prevent any overflow. The possible scenarios for data bit-width are 8 bits with 4-bit integers and 4 bits fractional, and 16 bits, with 8 bits integers and 8 bits fractional.

The metric that was used to determine whether weights should be represented as the 8-bit or 16-bit fixed points was estimated by using the weight loss function as given in equation 5.1. By calculating weight loss, one can estimate the impact of quantization without having to run the entire dataset. Weight loss is determined using a square law definition, where it represents the ratio of quantization noise power to signal power. This indicates the quantization effect on the network's weights. The weight loss function is calculated for 8-bit fixed point and 16-bit fixed-point representation. As each layer has a different weight distribution, some layers may be more sensitive to quantization than others. To investigate this, weight loss is measured for each layer, which is calculated as the ratio of quantization noise power

to signal power [38].

$$\text{weight loss} = \frac{\sum_{i,j}(w_{i,j} - \hat{w}i,j)^2}{\sum i,jw_{i,j}^2} \tag{5.1}$$

The quantization loss for 16-bit fixed-point representation is negligible as compared to 8-bit fixed-point representation as shown in table 5.3. This can also be supported by the histogram plot shown in appendix A.20. Therefore, the bit-width of weights is chosen to be 16 bits. To ensure consistency, both data and the weights are chosen to have the same width as the data, represented by 16-bit fixed-point numbers. Additionally, 32 bits are applied for the results of convolution windows to reduce the possibility of overflow and provide higher accuracy for accumulation. Please note that the distribution of the fixed point weight is obtained after combining batch-normalization operation with the floating point weights and then converted to fixed-point weights distribution. Batch normalization is mainly used for training purposes, not for inference, This batch normalization operation is combined with the weights optimization technique to speed up the execution of the convolutional layers.

layer	8-bit	16-bit
convolutional layer 0	7.24	0.28
convolutional layer 1	2.56	0.00
convolutional layer 2	3.57	0.00
convolutional layer 3	3.55	0.00
convolutional layer 4	4.55	0.00
convolutional layer 5	6.40	0.00
convolutional layer 6	7.30	0.00
convolutional layer 7	3.51	0.00
convolutional layer 8	6.75	0.00
convolutional layer 9	7.32	0.00
convolutional layer 10	7.42	0.00
convolutional layer 11	3.98	0.00
convolutional layer 12	24.36	0.00
convolutional layer 13	48.40	0.00
convolutional layer 14	17.85	0.00
convolutional layer 15	6.35	0.00
convolutional layer 16	1.36	0.00
convolutional layer 17	3.00	0.00
convolutional layer 18	60.30	0.00
convolutional layer 19	3.00	0.00
convolutional layer 20	40.50	0.00

Table 5.3: Weight loss: 8-bit vs 16-bit

The floating point model was modified to include the weights and data as fixed-point representations. The simulation was carried out for 16-bit fixed point representations of weight, data, and outputs. The computation time for the fixed point model is about 15 mins. It is important to note that while the software fixed-point simulation using `ap_fixed.h` is a useful tool, it cannot accurately replicate the behavior of the FPGA. This is because the `ap_fixed.h` library used in HLS is optimized for the hardware and is very slow when used in the software. Therefore, the simulation only performs integer shifting to approximate the behavior of the

`ap_fixed.h` library.

Additionally, to support the choice of choosing 16-bit over 8-bit fixed point representation was estimating the loss in accuracy which was computed using mAP for the model using the validation data set of 5000 images. The mean average precision of the 16-bit model is about 43.08% , as shown in listing 5.1 which is a loss of about 2% w.r.t floating point model mAP when validated with COCO Dataset 2017 Validation tests.

```
...
...
class_id = 53, name = pizza,   ap = 49.28 %
class_id = 54, name = donut,   ap = 60.55 %
class_id = 55, name = cake,    ap = 35.50 %
class_id = 56, name = chair,   ap = 27.10 %
class_id = 57, name = sofa,    ap = 41.09 %
class_id = 58, name = pottedplant,   ap = 30.17 %
class_id = 59, name = bed,   ap = 61.80 %
class_id = 60, name = diningtable,   ap = 25.88 %
class_id = 61, name = toilet,    ap = 74.76 %
class_id = 62, name = tvmonitor,   ap = 70.31 %
class_id = 63, name = laptop,    ap = 68.39 %
class_id = 64, name = mouse,   ap = 61.39 %
class_id = 65, name = remote,    ap = 11.64 %
class_id = 66, name = keyboard,    ap = 66.56 %
class_id = 67, name = cell phone,    ap = 33.54 %
class_id = 68, name = microwave,   ap = 61.60 %
class_id = 69, name = oven,    ap = 28.39 %
class_id = 70, name = toaster,   ap = 54.55 %
class_id = 71, name = sink,    ap = 57.45 %
class_id = 72, name = refrigerator,    ap = 35.95 %
class_id = 73, name = book,    ap = 10.89 %
class_id = 74, name = clock,   ap = 63.54 %
class_id = 75, name = vase,    ap = 24.59 %
class_id = 76, name = scissors,    ap = 4.55 %
class_id = 77, name = teddy bear,    ap = 49.75 %
class_id = 78, name = hair drier,    ap = 0.00 %
class_id = 79, name = toothbrush,    ap = 5.45 %

 for thresh = 0.25, precision = 0.72, recall = 0.33, F1-score = 0.45
 for thresh = 0.25, TP = 2310, FP = 883, FN = 4770, average IoU = 54.32 %

 mean average precision (mAP) = 0.430769, or 43.08 %
Total Detection Time: 26519.000000 Seconds
```

Listing 5.1: mAP for 16 bit fixed-point model

Chapter 6

Hardware IP Block Design

6.1 Motivation for using unified hardware architecture

To accommodate the different typology parameters of layers within a neural network, one option is to design specialized hardware for each layer. This approach can help reduce off-chip memory transactions and improve performance since each layer can be optimized for its specific task. However, a disadvantage of this approach is that the overall resource requirement for putting the entire network on hardware can be quite high.

An alternative approach to designing specialized hardware for each layer in a neural network is to separate the design into multiple-bit files. This means that the FPGA hardware needs to be reprogrammed when the network is forwarding, and once a layer is finished, the FPGA must be entirely reconfigured for the next layer. However, this approach can suffer from the problem of long reloading times for bit files, which could take hundreds of milliseconds, potentially exceeding the actual execution time. While the latency of each layer may be optimized using this approach, the overall performance of the network may suffer due to the overhead of reloading bit files.

This thesis uses a unified hardware architecture approach which is a compromise between the two strategies discussed above [1] [2]. A unified hardware architecture refers to a hardware design that efficiently handles different tasks or applications using the same set of hardware resources. This approach allows flexibility and adaptability to different use cases, without requiring significant changes to the hardware design. A unified hardware architecture that fits all layers of YOLOv4-tiny is a compromise solution to implement CNNs on FPGAs, especially when the intermediate results are too large to be stored on-chip Block RAM. This is useful in the case of implementing YOLOv4-tiny on resource-constrained devices like Zedboard which has limited on-chip memory of about 560Kb organized into 140 units with each unit capable of storing 2048x18 bit data. For instance, the sixth convolutional layer of YOLOv4-tiny utilizes an intermediate accumulator size of the $104 \times 104 \times 32$ words wherein 104x104 is the dimension of the output feature map and 32 is the number of channels that an IP can process. Each word is of size 16 bit due to reduced precision. The intermediate accumulator

size becomes equivalent to 0.66 MB. The design requires 2 such accumulators in case of input channel folding when the number of channels that need to be processed is more than the maximum number of channels that an IP can process. Therefore a total size of 1.22 MB will be required of on-chip RAM is required. This is more than what is available in ZedBoard. The concepts of input channel folding will be discussed in subsequent sections.

This approach allows for the *dynamic configuration* of CNN typological parameters at run-time, which can adapt to various CNN models and sizes [2]. typological features like input feature map dimensions, filter size, etc can be configured at the run time. The main advantage of this approach is its flexibility and compatibility not only with different layers in a CNN model like YOLOv4-tiny and also can be compatible with different CNN models itself like yolov3-tiny etc. The architecture can work under many settings by controlling the multiplexes inside, which allows for efficient use of hardware resources. The architecture presented in [2] uses a dynamically configurable architecture for Mobile robot vision applications using virtex4 FPGA running at 120 Mhz and achieves a latency of 25 ms. The architecture utilizes inter-output and intra-output convolutions for accelerating the layers of the CNN and each layer was parallelized in different ways. Dynamically configurable architecture provides speedups ranging from 1.2x to 3.5x over a similar fixed custom architecture for every layer. This thesis takes ideas of *dynamic configurability* for configuring each layer inside the PS section of the ZedBoard as it provides more flexibility for controlling the hardware but differs in the layers that are parallelized. The hardware is fixed for each convolutional layer and other layers after channel folding so the latency is more controlled by the topological features of the CNN.

Additionally, the architecture provides design parameters that can be tuned depending on the resource constraints of the target platform. By using a single hardware platform to handle different tasks or applications, a unified architecture offers better scalability than specialized hardware designs, which are typically limited to a specific function or application. The scalability of a unified hardware architecture makes it an attractive choice for systems that need to handle variable or unpredictable workloads. Additionally, The paper [1] uses ZedBoard as an indicative target platform but the design can be ported to target Xilinx FPGA devices with more resources due to its tunable design parameters. The design achieves optimal latency of 532 ms.

6.2 Convolutional Layer IP

In this section, the focus is on the convolutional IP's design, which is the most critical computationally intensive element in the system design. Section 6.2.1 provides the overview of the module which includes details of interfaces and other hardware components. Section 6.2.2 to 6.2.4 explain architectural details. Section 6.2.6 explains the optimization carried out in the Convolutional IP block and the directives applied in the design. The Explanation of architectural design details are taken from [1] and from the thesis [12]

6.2.1 Convolutional IP block design

Shown in Figure 6.1, are interfaces of the convolutional IP block which consists of data port interfaces and control ports. Data ports transfer layer weights, inputs, and outputs. To ensure fast data transfer due to a large amount of information, AXI4-stream ports are utilized [39]. Both weights and inputs are transmitted through the same 64-bit port, with weights transmitted first, followed by inputs. As for typology parameters, they are set using AXI4-Lite connections and each parameter has its own memory space in the PS.

In order to use the IP, certain parameters related to its typology must be sent to control logic first. Once these initial preparations have been completed, the weights needed for processing are transferred from the DDR (dynamic random-access memory) to the relevant local memory via an input stream. After this setup is finished, the input data is sent through the IP, which processes it and generates outputs that are then sent back to the DDR. This entire process forms a loop between the FPGA (field-programmable gate array) and the off-chip memory, with the processing system managing the flow of data.

To optimize the performance of convolution operations, multiple processing elements are

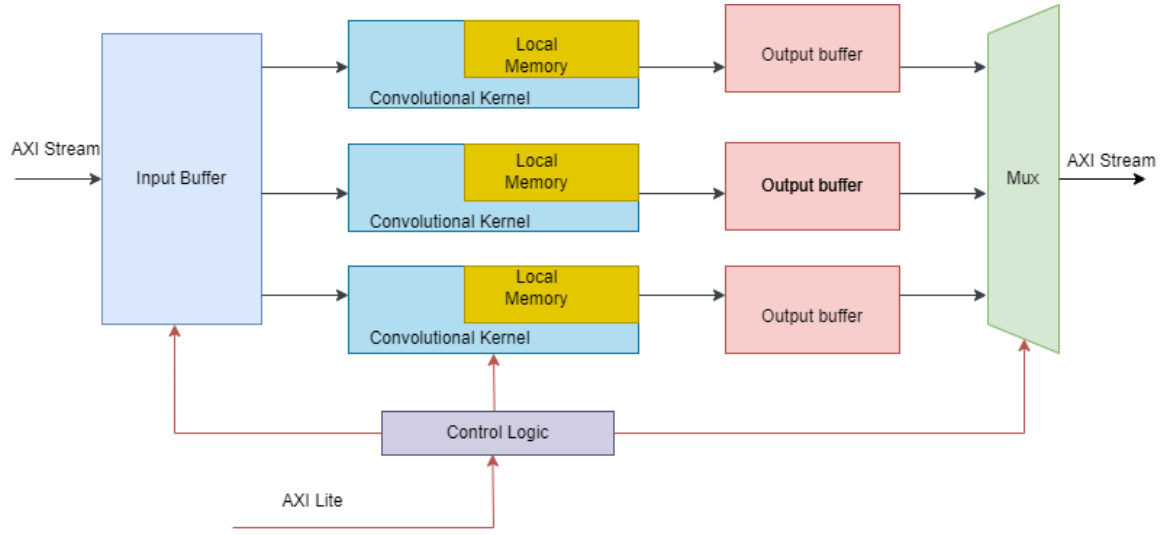


Figure 6.1: Structure of the convolutional IP [12]

available within the module that is capable of performing convolutions in parallel. This can provide significant hardware acceleration and improve the speed of processing. In addition, inputs and outputs should be buffered to allow for efficient data transfer between the processing elements. It is important to note that convolution is not a one-to-one mapping, which means that data will be reused multiple times during the operation. By buffering the inputs and outputs, the module can reuse data more efficiently and reduce the overall processing time. Proper optimization of the convolution module can lead to significant improvements in performance and efficiency. The input buffer is implemented using a tunable line buffer. Convolution across the channels is done using Multiply-Accumulate batch units. With each

batch unit, sliding windows are formed which take 3x3 size input from the line buffer and 3x3 size weight stored in local memory, and then outputs are buffers using output transfer blocks. All these are discussed in subsequent sections.

6.2.2 Tunable Line Buffer

Chapter 5 discusses the concept of partial buffering and full buffering. Full buffering refers to a technique where all inputs and outputs are stored in a buffer before any processing occurs. This can be useful when the data transfer rate between the FPGA and external memory is slower than the processing rate of the convolutional IP, allowing the IP to access data more efficiently. However, full buffering can also require a larger memory footprint, which may not be feasible in all cases. On the other hand, partial buffering is a technique where only a subset of the input and output data is stored in a buffer at a time, with the buffer being updated as the processing progresses. This can be more memory-efficient than full buffering but may also require more careful management of the data flow to prevent data loss or processing delays.

The original YOLO software program uses the Image to column algorithm for convolution, which may make it possible to optimize its performance using partial buffering (PB) on hardware. However, it is important to note that the Im2Col algorithm involves moving data in memory and making copies on the processor, which can be computationally expensive. In fact, when evaluated on the ARM processor of a Zedboard, the Im2Col functions of convolution took more than 5 seconds (2.72% of the total execution time of convolution layers) to complete as shown in Chapter 5 figure 5.8. This means that even with hardware acceleration, the system latency will never be lower than 5 seconds due to the computational cost of Im2Col. Therefore, to achieve lower latency, a Full buffering (FB) strategy is adopted.

In the FB method, the input image is processed by sliding a convolutional window over it in row-major order. As the window moves, only recent rows are required to be stored in memory, which means that the FPGA doesn't need to keep every input in memory all the time. This approach allows for efficient use of memory and maximizes data reuse, resulting in faster processing times. To achieve this, the design uses a line buffer structure shown in 6.2, which is essentially an array of shift registers that store a single row of input data at a time. The size of the line buffer depends on the height of the convolutional kernel, with each channel having a buffer of k_h lines, where k_h is the height of the kernel, and each line contains f_w words, where f_w is the width of the input image. By using a line buffer structure, the design can store input data more efficiently and minimize the memory required to perform convolutions.

The line buffer structure acts as a shift register array that only stores the most recent rows of data. When a new input is captured, the data in the corresponding column is shifted up and the new input is inserted. The size of the convolutional kernel and the dimensions of the

input determines the line buffer size. In the YOLOv4-tiny typology, the maximum values for the kernel size and input dimension are 3 and 416, respectively. However, the line buffer has 3 lines and 418 words for each row to account for convolution padding. Storing padded zeros avoids the need for extra multiplexers and simplifies the control logic, which reduces the risk of timing violations. By trading a small amount of memory for simpler control logic, the overall system can achieve better performance and efficiency.

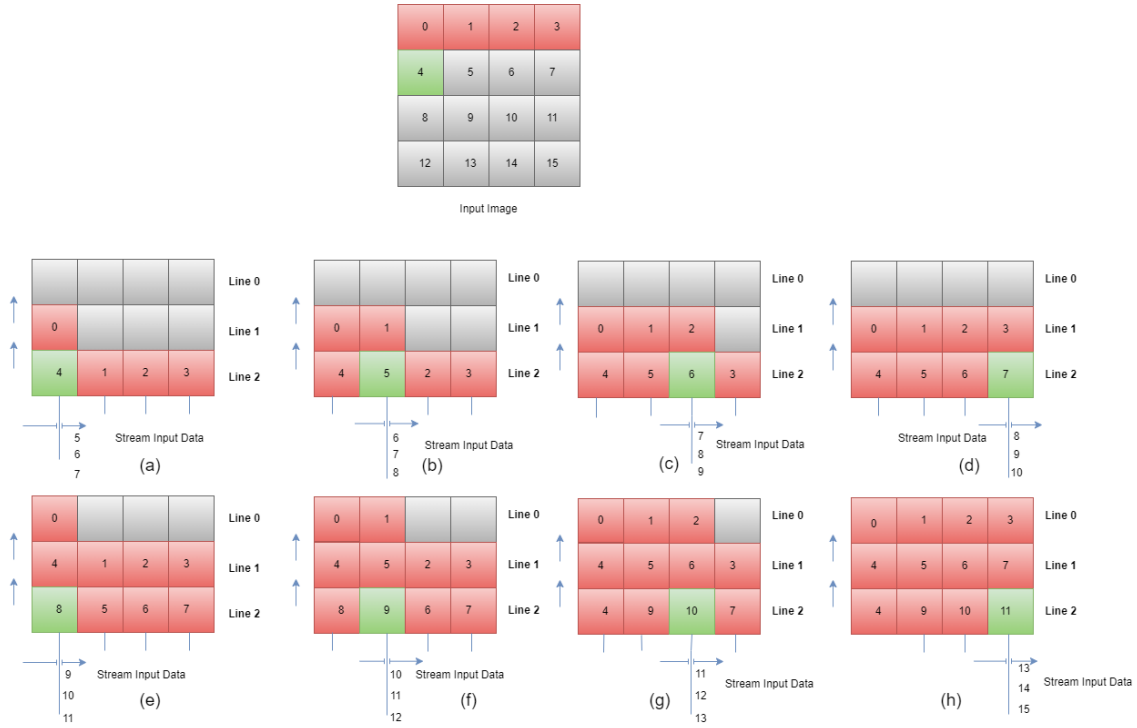


Figure 6.2: Data movement inside line buffer

6.2.3 3x3 Sliding Window

The sliding window technique is used to retrieve data from the buffers and initiate convolutions at the appropriate cycle. Figure 6.3 shows an example of a 3x3 sliding window function. When the convolutional kernel is 3x3, the first convolution will occur once the top two lines and the first three columns are filled with data 6.3 (A, B). Subsequently, for each new row, the convolution will not start until at least three new elements are fetched. This ensures that the convolution is performed on a complete set of data for accurate results 6.3 (C, D). The sliding window approach is an efficient way to manage the data flow and timing in the convolution process, resulting in optimized performance and reduced latency.

6.2.4 Multiply Accumulate batch units

The main objective of the unit is to perform convolutions, which involve two main steps. The first step is the window convolution, which can be seen as an inner product of two vectors. In the case of a 3 x 3 convolution kernel, it includes 9 multiplications and 8 additions shown

Figure 6.3: Sliding window for a 3x3 convolution

in figure 6.4 as p_{c3}. This is called intra-FM convolutions. The second step involves adding up the convolution results of all input channels, 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 N_{in} times. Parallelism is therefore achieved among different output channels as by p_{c1}. In this design, inputs belonging to four input channels will be transferred in parallel. This is because the bit-width of data is a quarter of the 64-bit DMA. The code specifies that every multiply-accumulate unit must include four convolution kernels to calculate four input channels simultaneously given by p_{c2} in figure 6.4. This is also called inter-FM convolution. In order to minimize resource consumption, these kernels can share one accumulation unit as shown in Figure 6.4. This entire multiply-accumulate unit is referred to as a "multiply-accumulate batch" for the sake of clarity. When there are N_{MAC} multiply-accumulate batches, all N_{MAC} output channels and four input channels will run in parallel. This will result in a total speed-up of 4 x N_{MAC}, when compared to sequential execution.

6.2.5 Output stream merge

Within the module, multiple output channels are operating concurrently. The convolution process accumulates data for N_{in} input channels, and outputs become available after processing the last input channel. In a pipeline setup, N_{out} output channels will be generated simultaneously when N_{in} input channels are consumed. This means that data write operations will occur more frequently than reads, if N_{out} is greater than N_{in}. As a result, there may be data congestion at the output port.

As per the AXI4-stream protocol, each port is allowed to have only one read and one write transaction during a single clock cycle. In the current system where the DMA port is config-

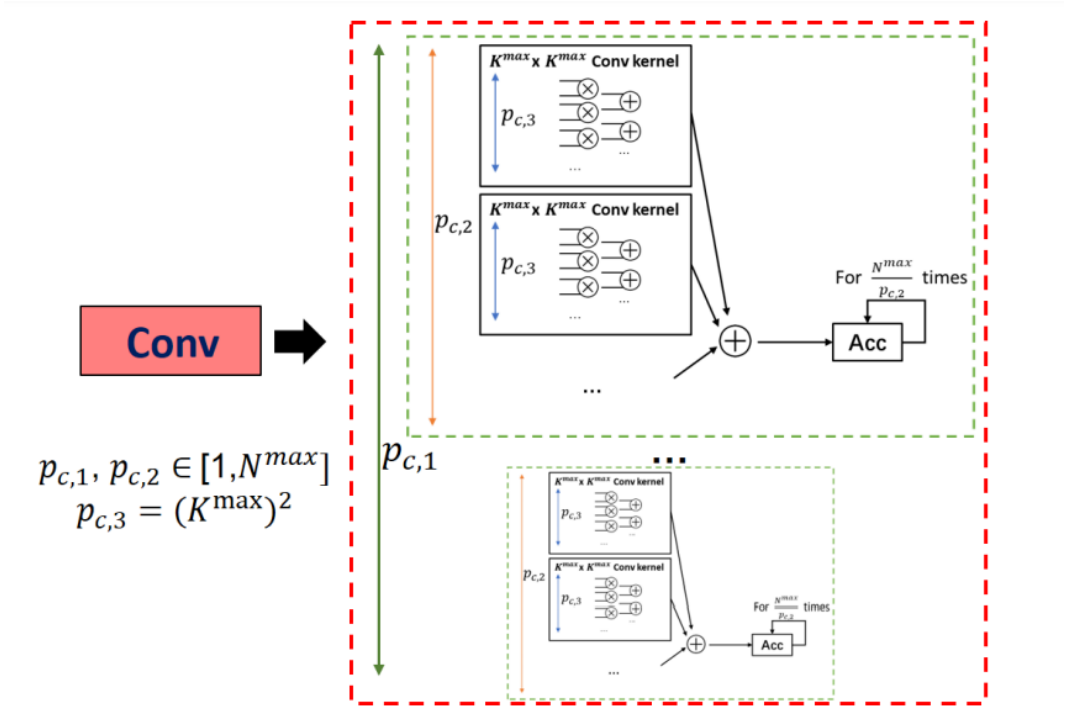


Figure 6.4: Multiply Accumulate Batch Units [1]

ured as 64 bits and each fixed-point data occupies 16 bits, only 4 channels can be read and written at a time. If the number of output channels (N_{out}) is larger than the number of input channels (N_{in}), it may result in pipeline stalling as writes need to finish before new data can be processed. This is because the system has to wait until all writes are completed before proceeding with further operations, leading to potential data congestion at the output port. This is a deterministic behavior of the AXI4-stream protocol. This is because the protocol is designed to allow only one write transaction per port per clock cycle, which means that if there are multiple write transactions, they have to be serialized and completed before the next set of transactions can proceed.

One possible solution to resolve the conflict is to add a FIFO (First-In, First-Out) buffer between the convolution outputs and the AXI4-stream port. This buffer can help manage the flow of data and prevent congestion at the output port. For example, let's consider a convolutional layer with 8 input channels and 16 output channels, where the AXI4-stream port is set as 64 bits as shown in figure 6.5. If the initiation interval of the module is 1, it takes 2 cycles to fetch all the needed inputs. Assuming there is no latency between inputs and outputs, in the second cycle, 16 outputs are waiting for transmission. By adding a FIFO buffer, the outputs can be stored temporarily in the buffer until they can be transmitted through the AXI4-stream port without causing congestion. This can help maintain smooth data flow and prevent stalls in the pipeline. The registers that store output values will be cleared when the next 8 input channels arrive, causing the input stream to stall for three extra cycles. As a result, the entire read-and-write process takes 5 cycles to complete. Essentially,

the purpose of these output FIFOs is to pipeline the stream read and stream write at the task level.

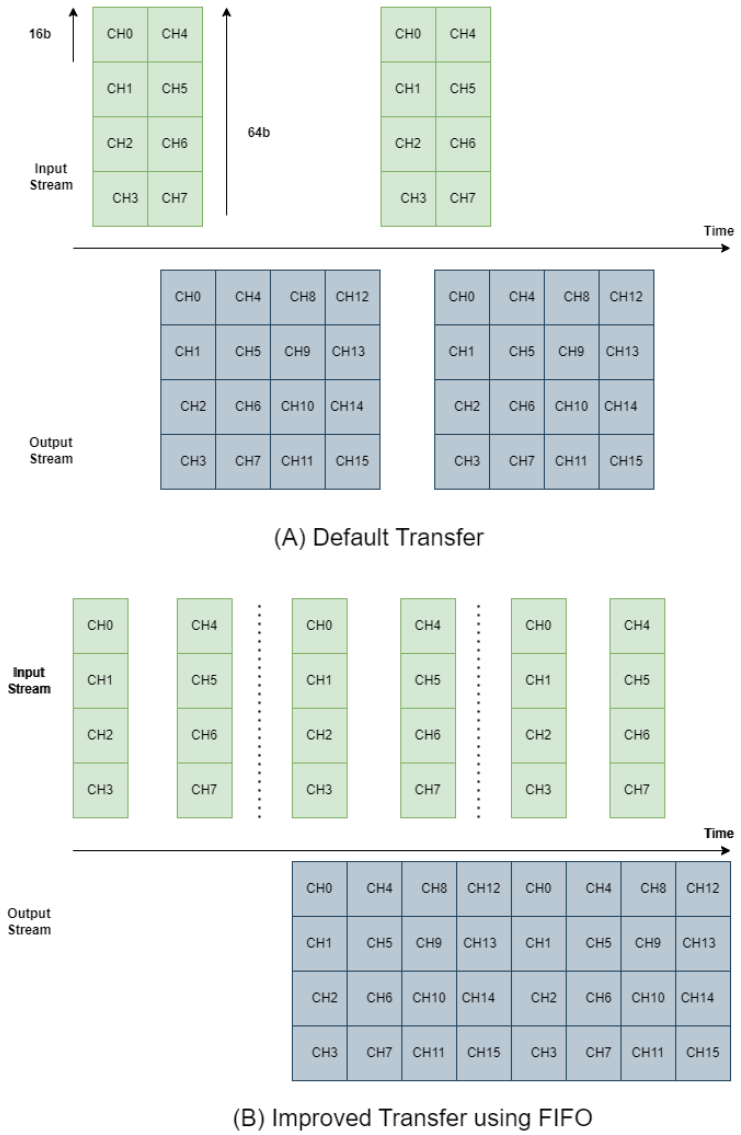


Figure 6.5: Merging of output stream [12]

6.2.6 Optimisation

6.2.6.1 Merging Batch Normalisation

As explained in Chapter 2, YOLOv4-tiny incorporates batch normalization after most of its convolutional layers, followed by activation. In batch normalization, we need to store and transfer four parameters per output channel: scale, bias, mean, and variance. Since there are N_{out} output channels, the total number of parameters needed is $4 \times N_{out}$. These parameters are used to normalize the activations of each output channel to have zero mean and unit variance, which helps with the stability and speed of the training process. To perform batch normalization according to equations 2.3 and 2.4, a total of $4 \times N_{out}$ parameters need to

be transferred and stored on the FPGA. This operation also requires 2 x g_h x g_w x N_{out} multiplications and 2 x g_h x g_w x N_{out} additions.

The FPGA design under consideration is not meant for training but only for inference. Thus, it is acceptable to eliminate batch normalization using mathematical methods. By replacing g_i in equation 2.3 with equation 2.4, we can obtain a new equation without the need for batch normalization.

In order to avoid the cost of transferring and storing $4 \times N_{out}$ parameters for batch normalization, a mathematical technique is used to merge the operations of batch normalization into convolution. This is achieved by introducing new weights and biases, $w'_{i;j}$ and b'_j, which can be computed before run-time with negligible precision losses during floating point calculations.

$$g'_j = \frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} g_j + \beta - \frac{\gamma \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{6.1}$$

Wherein g_i comes in from equation 2.1, where b_j is 0. Therefore,

$$g'_j = \sum_{i=1}^{N_{in}} f_i * \left(w_{i,j} \frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \beta - \frac{\gamma \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad \text{with} \quad j \in [1, N_{\text{out}}] \tag{6.2}$$

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

$$g'_j = \sum_{i=1}^{N_{in}} f_i * w'_{i,j} + b'_j, \quad \text{with} \quad j \in [1, N_{\text{out}}] \tag{6.3}$$

6.2.6.2 Channel Interleaving

Three distinct methods exist for transferring data between FPGA and off-chip memory. As per equation 2.1, the convolution operators necessitate accumulation over all input channels. Consequently, a simple approach involves transmitting all the pixels in one input channel in sequence before switching to the next channel. This is depicted in Figure 6.6. For instance, let's consider the first convolutional layer that has RGB inputs with an input width and height of 416 pixels. In this case, the 416 x 416 pixels in the "R" channel are transferred first, followed by the 416 x 416 pixels in the "G" channel, and then the 416 x 416 pixels in the "B" channel. However, this method can lead to a problem with the accumulation buffer. It can be explained as follows. During the convolution operation, each output channel accumulates results from all input channels. Therefore, a temporary output buffer is required to store these values. The size of this buffer is determined by the layer outputs' dimensions ($g_h \times g_w$). As the IP is a unified module, the buffer must be designed for the worst-case scenario, which is 416 x 416 x the word length (in bits). However, this output buffer's size is too large to fit on the test device (Zynq-7020). Implementing the buffer using off-chip memory would result in significant delays.

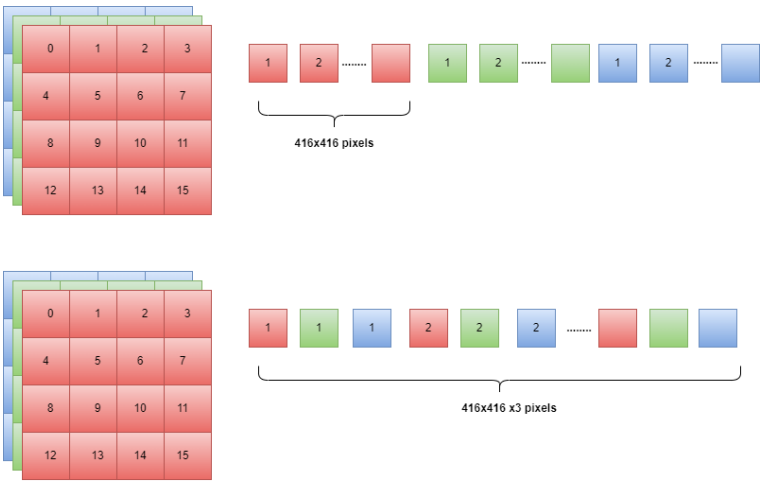


Figure 6.6: Sequential vs Channel Interleaving transmission

The second solution involves interleaving channels as shown in Figure 6.6 (b). In this approach, the pixels are sent in a certain order where the first pixel from the "R" channel is sent, followed by the first pixel from "R" and the first pixel from "B" channels, and so on until the last pixel is reached. This process transposes the original 3D matrix mathematically. With interleaving, the accumulation process occurs for a single pixel rather than the entire channels, which reduces the size of the output buffers.

While channel interleaving provides a solution to the accumulation buffer problem, it comes at a cost. Unlike sequential channel transmission, which only requires one line buffer to store data of one input channel, channel interleaving requires separate line buffers for each input channel. This is depicted in Figure 6.7 However, this also means that the total buffer size is determined by the maximum number of input channels that the IP can handle. This is a desirable feature because the buffer size can be adjusted as a design parameter.

In summary, the sequential channel transmission approach requires a large and fixed output buffer, while channel interleaving requires multiple input buffers. However, the adjustable buffer size feature makes channel interleaving a more attractive option.

The third option is to combine the previous two strategies by dividing each input channel into batches, and interleaving the batches instead of individual pixels. This approach strikes a balance between the previous methods, but it still requires moving and copying data on the processor, which is not efficient for the processing system (PS) of the test platform used in this project.

6.2.7 Architecture of the convolutional IP

Listing 6.1 shows the algorithm for calling the convolutional IP. Here N_{out} and N_{in} are the output channels and input channels of a convolutional layer whose values are always equal or

less than the maximum channels that can be processed at a time N_{max}. In this design, N_{max} is set to 32. The concept of network transformation as discussed in chapter 7 is applied where in the IP is called multiple times depending upon the number of input and output channels of a convolutional layer. For example, In YOLOv4-tiny, the third layer is a convolutional layer input size of $104 \times 104 \times 64$ with 64 filter weights and output size of $104 \times 104 \times 64$. While performing convolutions, all the input channel data is used to compute a single output feature map. Due to the constraint of processing 32 channels at max in a single call of IP, computing the first 32 output channels (N_{out}) will require all 64 channels inputs (N_{in}) but input processing has channel constraints of processing a maximum of 32 channels. So input needs to be folded twice for processing the first 32 output channels. Similarly, the next 32 output channels require the same concept. Therefore in total, this IP will be called 4 times for this layer execution. For clarity, all layers except the first convolutional layer take (N_{out}) and (N_{in}) values to be 32 and 32 after network transformation even though the original values of (N_{out}) and (N_{in}) are 64 and 64.

```
//Loading weights inside local memory
Loop 1: for i = 0; i < Nout; i++ do
            for j = 0; j < Nin; j++ do
                for k = 0; k < 3; k++ do
                    #pragma HLS PIPELINE II=1
                    load weight;
                end
            end
        end
// Convolutional Operation
Loop 2: for i = 0; i < fh + 1; i++ do
            for j = 0; j < fw; j++ do
                for k = 0; k < Nin; k++ do
                     #pragma HLS PIPELINE II=IIconv
                     line buffer;
                     sliding window function;
                     conv mac;
                     output stream merge;
                end
            end
        end
```

Listing 6.1: Algorithm of Convolution IP

Figure 6.7 shows the detailed architecture of the convolution IP. As discussed earlier, the line buffer size chosen for this design for storing input is 3×418. The height of the line buffer size, as discussed earlier is chosen based on the 3x3 kernel dimensions and the width of the line buffer is based on 1^{st} convolutional layer whose dimensions are 416×416. In the module for convolution, there are two nested loops: one for loading weights and the other for performing the convolution as shown in figure 6.1. Since there is a dependency between the iterations of the second loop due to accumulation, the *PIPELINE* directive has been used.

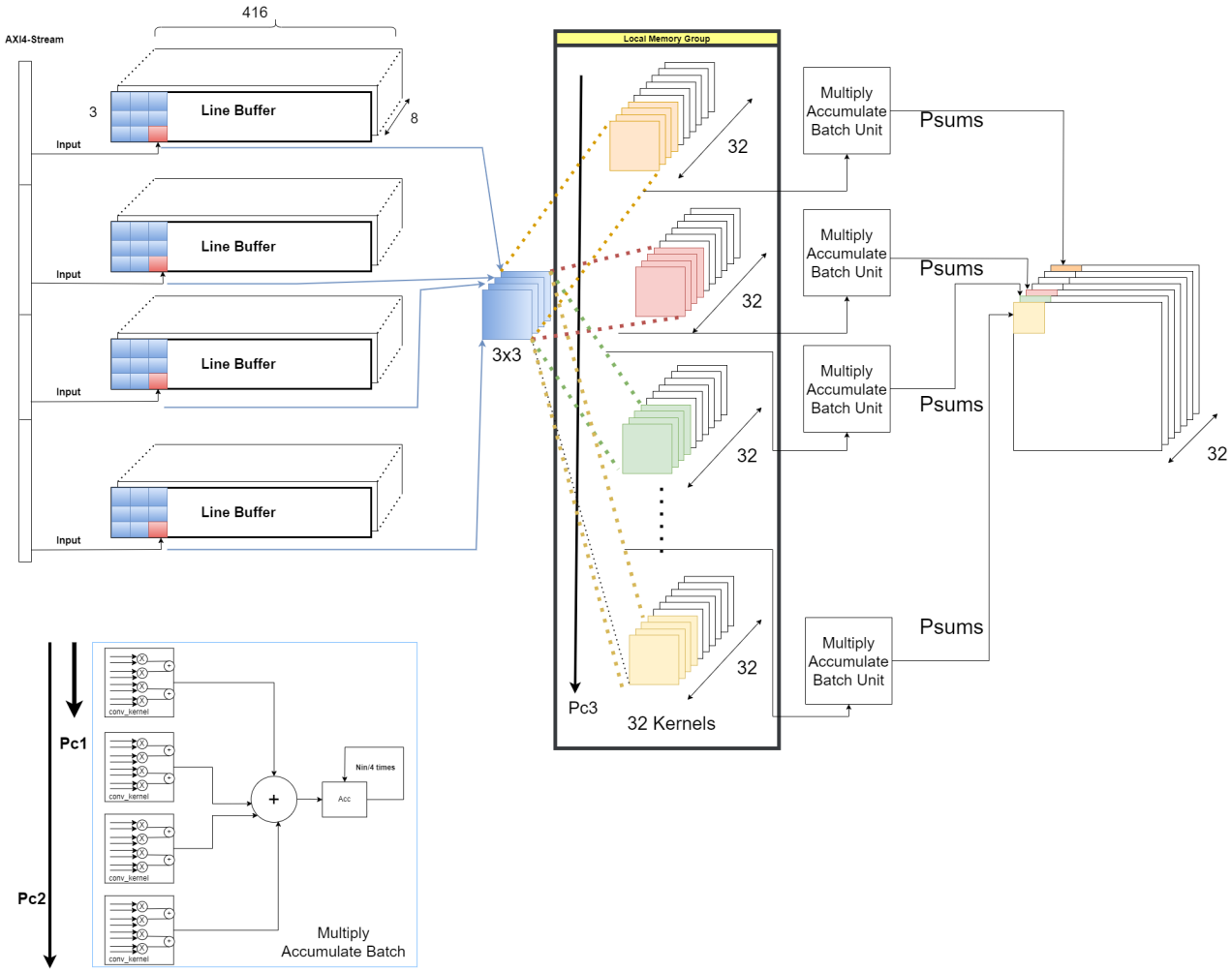


Figure 6.7: Detailed architecture of convolutional IP

Secondly, *Array Partition* directive has been used to partition the weights stored inside the local memory buffer. The weights of a convolutional layer have a specific size, which can be represented as a 3-dimensional array with dimensions $(N_{out}, N_{in}, k_h * k_w)$. The challenge is to determine how to partition this array and understand how that partitioning will affect the design.

The way the 3-D array of weights is partitioned affects the performance of the convolutional layer. The number of blocks in the first dimension of the array corresponds to the number of output channels that can be processed in parallel which is shown by (P_{c3}) in figure 6.7. As local memories are implemented using 2-port BRAMs, then having P_{mem1} blocks in the third dimension will enable $P_{mem1} \times 2$ convolutional kernels or a total of $\frac{p_{mem1}}{2}$ multiply-accumulate batches will be executed concurrently. If the number of desired output channels N_{out} is greater than $\frac{P_{mem1}}{2}$, some multiply-accumulate batches will have to be reused, leading to pipeline stalls. Thus the number of memory ports determines the number of multiply-accumulate instances and ultimately affects the total latency of the convolutional layer.

The second dimension of the 3D weight array corresponds to the different input channels, which can be divided into P_{mem2} blocks using cyclic partitioning which is shown by (P_{c2}) in figure 6.7. However, for the 64-bit DMA transfer, only four input channels can be processed in parallel at most. Therefore, if P_{mem2} exceeds 4, no further speed-up can be achieved and the extra resources are wasted.

The third dimension of the weight array in a convolutional layer is related to the multiply-accumulate (MAC) batch that is completely unrolled within the pipeline which is shown by (P_{c1}) in figure 6.7. For example, in a 3x3 convolution, 9 multiplications should be scheduled in the same cycle if the memory access allows it. However, if the unrolled units are not used efficiently, the throughput of the batch will decrease. Therefore, in the design process, the third dimension is always fully partitioned to ensure efficient usage of the unrolled units.

In summary, the first and second dimensions control the number of instances, while the third dimension determines the performance of each instance. If the third dimension is fully partitioned, it can maximize the throughput of the instance. The overall latency of the pipeline is affected by all three dimensions. The way the array partition needs to be carried out is discussed in chapter 4

Figure 6.8 shows the variation of Initiation interval cycles with respect to memory partitioning (p_{mem}) wherein $P_{mem} = P_{mem1} \times P_{mem2}$ with $P_{mem2} = 1$ for this design as 4 channels can be parallelized without partitioning the second dimension. It can be clearly seen that the initiation interval of the pipeline reduces as more memory partition is increased. $P_{mem} >= 8$ causes the initiation interval to be 10 clock cycles. So less latency is expected when $p_{mem} >= 8$. Furthermore from table 6.1, it can be seen that the more efficient utilization of DSP happens when $P_{mem} >= 8$ and resource utilization increases with no change in the initiation interval of the pipeline. Thus $P_{mem} = 8$ is selected as the more optimal value for this design.

DPS usage of the IP block depends upon a number of kernel instances that can be parallelized which again depends upon how on memory port or how memory has been partitioned. During one iteration of the nested loop, it has been already discussed that $4 \times N_{max}$ operations can be parallelized. So the number of kernel instances inferred by the design will be $\left\lceil \frac{N_{max} \times 4}{II_{conv}} \right\rceil$. Each instance does 3×3 window convolution, so 9 DSP cores will be inferred by each instance. Additionally, 2 more DSP cores have been inferred by the design as control logic of the code. Therefore in total DSP cores inferred by the design will be given by:

$$\mathrm{DSP_{conv}} = \left\lceil \frac{N_{max} \times 4}{II_{conv}} \right\rceil \times 9 + 2 \tag{6.4}$$

As N_{max} for the whole design has been set to 32. Therefore design infers 119 DSP units out of 220 available on Zedboard. The design inferred 13 instances of Multiply accumulate batches in each of which 9 multiply-accumulate operations were done in parallel. Each operation takes about 20 ns which is 2 clock cycle as the design frequency is 100MHz as shown in figure 6.9.

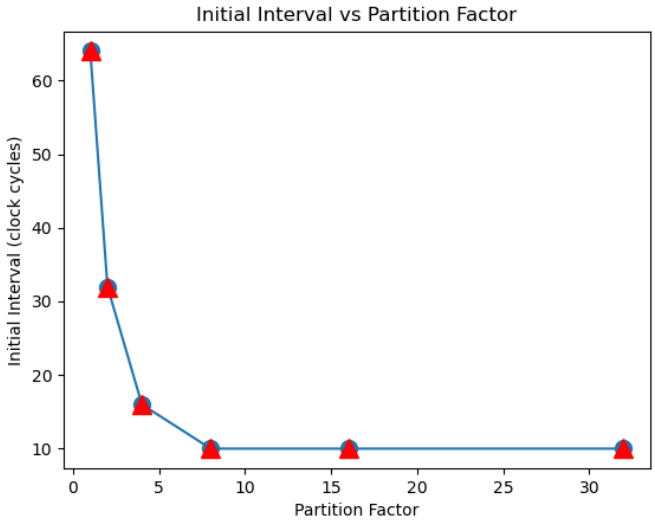


Figure 6.8: Initiation interval cycles variation with Memory partition for $N_{max} = 32$

Therefore the IP gives a throughput of $\frac{13\times9}{2} \times 100 = 5.85$ GOPS/sec. BRAM utilization

```
+ Detail:
    * Instance:
    +-------------------------------+------------------+---------+---------+-----------+-----------+-----+-----+----------+
    |                               |                  | Latency (cycles) |   Latency (absolute)  |  Interval | Pipeline |
    |           Instance            |      Module      |   min   |   max   |    min    |    max    | min | max |   Type   |
    +-------------------------------+------------------+---------+---------+-----------+-----------+-----+-----+----------+
    |grp_out_stream_merge_fu_11954  |out_stream_merge  |        8|        8| 80.000 ns | 80.000 ns |    8|    8| function |
    |grp_window_macc_fu_12012       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12034       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12056       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12078       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12100       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12122       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12144       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12166       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12188       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12210       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12232       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12254       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_window_macc_fu_12276       |window_macc       |        2|        2| 20.000 ns | 20.000 ns |    1|    1| function |
    |grp_slide_window_fu_12298      |slide_window      |        3|        3| 30.000 ns | 30.000 ns |    3|    3| function |
    |grp_slide_window_fu_12307      |slide_window      |        3|        3| 30.000 ns | 30.000 ns |    3|    3| function |
    |grp_post_process_fu_12316      |post_process      |        0|        0|    0 ns   |    0 ns   |    1|    1| function |
    |grp_post_process_fu_12327      |post_process      |        0|        0|    0 ns   |    0 ns   |    1|    1| function |
    |grp_post_process_fu_12338      |post_process      |        0|        0|    0 ns   |    0 ns   |    1|    1| function |
    |grp_post_process_fu_12349      |post_process      |        0|        0|    0 ns   |    0 ns   |    1|    1| function |
    +-------------------------------+------------------+---------+---------+-----------+-----------+-----+-----+----------+
```

Figure 6.9: Latency of Multiply accumulate batch [12]

again depends on how memory has been partitioned. This BRAM estimation in the design can be made using BRAM utilization estimation in [12]

$$\text{BRAM}_{\text{conv}} = \left\lceil \frac{N_{max}}{8} \right\rceil \times 12 + \left\lceil \frac{N_{max}^2}{1024 \times P_{mem}} \right\rceil \times P_{mem} \times 9 \tag{6.5}$$

6.2.8 Latency Estimation of Convolutional IP block

The latency of the convolutional IP block can be estimated once the topological parameters of the design are known. This is because the bitstream generated by the tool gets fixed once the maximum number of channels processed by IP gets fixed. In the IP block, the number of input channels (N_{in}) and number of output channels (N_{out}) after folding is kept same as

	$p_{mem} = 1$	$p_{mem} = 2$	$p_{mem} = 4$	$p_{mem} = 8$	$p_{mem} = 16$	$p_{mem} = 32$
DSP48E	20 (9 %)	38 (30%)	74 (33 %)	119 (54 %)	119 (54%)	119 (54 %)
FF	11114(10 %)	17871 (16 %)	29247 (27 %)	37680 (35 %)	40095(37%)	50938 (47 %)
LUT	19253 (36 %)	22263 (41 %)	26438 (49 %)	28601(53 %)	30638(57%)	34556 (64 %)
BRAM	57(20 %)	66 (23 %)	84 (30 %)	120(42 %)	192 (68%)	48 (17 %)

Table 6.1: Resource utilization variation with memory partition (P_{mem}) for $N_{max} = 32$

maximum number of channel that an IP can process (N_{max}). As a result, IP block latency is governed by the software driver. This transfers IP block topological information like the number of desired input channels, the number of desired output channels, and feature map height, and width through the AXI interface.

The reason for keeping the number of input channels and the number of output channels the same as the maximum number of channels that IP can process is to avoid channel re-interleaving in case of multiple executions of the same IP block. Additionally, the interleaving channel requires data movement of data in the PS which has been proven to be bad as it takes more than 5 secs to move data. Therefore, the latency of convolutional IP is given by:

$$\text{Latency}_{\text{conv}} = P_{\text{conv}}\ (II_{\text{conv}}, \mathbf{N})$$

where II_{conv} is the initiation interval of the convolutional block pipeline and N is the topological features given by

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

The II_{conv} for the IP block is found to be 10 clock cycles after synthesizing the design. According to the algorithm given in 6.1, there are 2 loops, one for transferring weights and one for doing convolutions. The 3x3 weights are requires 3 clock cycles and an initiation interval of 1 is set which was easy to achieve for this loop. This 3x3 weight transfer has to happen $N_{out} \times N_{in}$ times. So estimated latency in clock cycles of the weight transfer loop is

$$\text{Latency}_{\text{weights}} = N_{\text{in}} \times N_{\text{out}} \times 3 \tag{6.6}$$

The second loop latency is related to the trip count of the second loop which is also related to topological parameters f_h, f_w, N_{in}. Inside the loop, the values of f_h and f_w are incremented by 2 to account for padding in the YOLOv4-tiny model. All Convolutional layers in YOLOv4-tiny require padding. One extra row is also sent as shown in fig 6.1 to take into account the delay of one output pixel for allowing the read-write pipeline. Also due to the DMA constraint of processing 4 channels in parallel a fixed factor of 4 is taken into account. Considering all this, the latency of the second loop is given by

$$\text{Latency}_{\text{op}} = (f_{\text{h}} + 3) \times (f_{\text{w}} + 2) \times \left\lceil \frac{N_{in}}{4} \right\rceil \times II_{\text{conv}} \tag{6.7}$$

So total latency in clock cycles is

$$\text{Latency}_{\text{conv}} = \text{Latency}_{\text{weight}} + \text{Latency}_{\text{op}}$$

$$\boxed{\text{Latency}_{\text{conv}} = (f_{\text{h}} + 3) \times (f_{\text{w}} + 2) \times \left\lceil \frac{N_{in}}{4} \right\rceil \times II_{\text{conv}} + N_{\text{in}} \times N_{\text{out}} \times 3} \quad (6.8)$$

6.3 Accumulation & Activation IP block

In some cases, the maximum number of input channels (N_{max}) that can be processed simultaneously by the convolutional IP is smaller than the desired number of input channels (N_{in}). This means that the convolutional IP cannot accumulate over all input channels through one execution, and multiple launches are required to process all the input channels. To overcome this issue, a separate module is designed that adds the results of all launches together. This module is responsible for folding the input channels and accumulating their results, which is why this technique is called input channel folding. The main idea behind input channel folding is to split the input channels into multiple groups that can be processed by the convolutional IP simultaneously. After all groups are processed, their results are accumulated to produce the final output.
The accumulation and activation module, shown in figure 6.10 is designed to add the results of multiple launches of the convolutional IP in cases where N_{max} is smaller than the desired N_{in}. It has two input streams, one for the output of the convolutional IP and the other connected directly to the PS to fetch outputs from previous launches. The module accumulates the results from both streams and applies biases stored in local memory through stream B. The output is then activated using either linear or leaky ReLU activation functions. For implementing leaky ReLU, fixed-point multiplication is required. The module also has a bundled AXI4-Lite port for setting parameters.

In order to optimize the resource utilization and latency, a tunable parameter P_{acc} is used which cyclically partitions the bias memory as shown in figure 6.10. Due to DMA and 16-bit quantization constraints, accumulation only can happen parallelly across 4 output channels. But due to limited memory ports (2-Port BRAM), there was a need to partition the memory cyclically for allowing access to biased memory in a single clock cycle. As shown in table 6.2, the initiation interval of the pipeline is reduced to 1 clock cycle when bias memory is partitioned by a factor of 2. This allows simultaneous access to bias memory thereby reducing the initiation interval of the pipeline. Value of $P_{acc} > 2$ causes the memory to be partitioned unevenly as memory can hold only 32 bias values. This uneven partition causes the HLS tool to infer more LUTs as compared to BRAM. In terms of DSPs, 4 DSPs come from the parallel processing of 4 channels which involve leaky activation to be applied across 4 channels. Additional DSP comes from the control logic inside the code. Therefore, considering resource utilization and latency $P_{acc} = 2$ is a more optimal value and is chosen for this design.

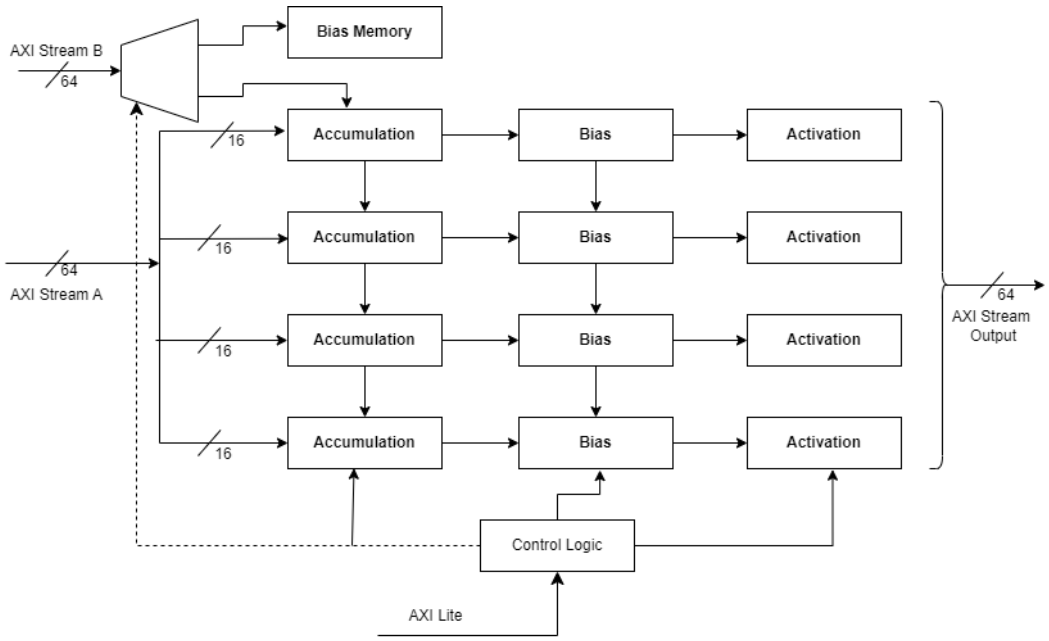


Figure 6.10: Accumulation IP block

	$p_{acc} = 1$	$p_{acc} = 2$	$p_{acc} = 3$	$p_{acc} = 4$
DSP48E	3 (1 %)	5 (2%)	5 (33 %)	5(2%)
FF	1234 (1%)	1387 (1%)	2718 (2%)	1515 (1%)
LUT	1930 (3%)	2284 (4 %)	3240 (4)	2232(4%)
BRAM	1(0 %)	2 (0)	0 (%)	0(0%)
Initiation interval (II_acc)	2	1	1	1

Table 6.2: Latency and resource utilization for various values of P_{pool}

6.3.1 Latency estimation of Accumulation block

As explained earlier, biases are transferred through the DMA stream B interface. Due to the DMA constraint of processing 4 channels at a time, the number of clock cycles that will be required for transferring biases will be:

$$\text{Latency}_{\text{bias}} = \left\lceil \frac{N_{out}}{4} \right\rceil \tag{6.9}$$

The rest of the process is accumulation which happens inside for loop whose latency is given by which by the trip count of the loop and the initiation interval of the accumulation pipeline block (II_{acc}). Loop trip-count is the product of the g_h, g_w, and N_{out}.

$$\text{Latency}_{\text{op}} = g_{\text{h}} \times g_{\text{w}} \times \left\lceil \frac{N_{out}}{4} \right\rceil \times II_{\text{acc}} \tag{6.10}$$

So total latency in clock cycles is

$$\text{Latency}_{\text{acc}} = \text{Latency}_{\text{op}} + \text{Latency}_{\text{bias}}$$

$$\boxed{\text{Latency}_{\text{acc}} = g_{\text{h}} \times g_{\text{w}} \times \left\lceil \frac{N_{out}}{4} \right\rceil \times II_{\text{acc}} + \left\lceil \frac{N_{out}}{4} \right\rceil} \tag{6.11}$$

6.4 Max Pooling Layer IP block

P_{pool} is the tunable design parameter for the Maxpool layer which controls the hardware for making comparisons across the channels using the HLS_ALLOCATION pragma. Again due to DMA constraints, only 4 channels can be parallelized. So the choice of the value of P_{pool} is determined by a trade-off between latency and resource utilization. Table 6.3 provides the hardware resource utilization with respect to various values of P_{pool}. The FF utilization is approximately 1% and LUT utilization is between 4% and 5%. The initiation interval will depend upon the P_{pool} values and the time it takes to access the line buffers. For instance, $P_{pool} > 1$ means there a separate instance of the comparison hardware will be used across 4 channels which will lower the initiation interval of the pipeline to 1 but accessing the line buffer in this design takes 2 clock cycles irrespective of the value of $P_{pool} = 4$. So line buffer access time is the lower bound time in clock cycles which determines the initiation interval in case of $P_{pool} > 1$. For the case of $P_{pool} = 1$, one such hardware instance is shared across 4 parallel channels and this will cause pipeline depth to increase by 4 clock cycles. In this case, extra clock cycles spent resource sharing becomes a more dominant factor as compared to the access time of line buffer which makes the initiation interval 4.

Latency in the table here is for a single call of IP which means latency of the IP here is with respect to processing 32 channels which are in accordance with the maximum channel that an IP can process in a single call. As there are 3 Maxpool layers with different numbers of output channels, the concept of folding will be applied which will cause latency to be multiplied by the fold factor determined by the number of output channels. The latency will be dependent on the typological features of the layer which decides the number of times the loop needs to be executed and the folding factor. The resource utilization remains the same in each call of IP. For example, Layer 10 in YOLOv4-tiny is a Max pool Layer with 128 output channels and input dimension as 104. So a factor of 4 needs to be taken into account for calling the IP 4 times which makes the choice of $P_{pool} = 1$ less efficient as Hardware latency will be $3.462 \times 4 = 13.848$ as compared to the cases when $P_{pool} > 1$ where latency becomes 6.92 ms. So taking into consideration almost the same resource utilization but varying latency, a more optimal choice was to set $P_{pool} = 2$ for this IP block.

The line buffer data structure which will utilize BRAM is used again in this IP with a size of 2×104. The reason for setting the height as 2 is because max-pool requires a window of size of 2×2. Only 2 rows are required for selecting the largest element. The width is set to the maximum width of the input image in YOLOv4-tiny. There are only 3 Maxpool layers in YOLOv4-tiny and the maximum width dimension of the Maxpool layers comes from the 10th layer of YOLOv4-tiny which has an input dimension of 104×104. As the maximum

number of channels that IP can process is set to 32, therefore the number of words stored inside the line buffers are $2 \times 104 \times N_{max}$. Due to Bandwidth constraints, only 4 channels can be processed parallelly, which means there are 8 groups of line buffers that can store $104 \times \lceil \frac{N_{max}}{4} \rceil$ words. Suppose there are only BRAM18k available for use, with each BRAM capable of storing 1k words of 16-bit data. However, it's important to note that BRAM18k in Xilinx devices can only support 1, 2, 4, 9, or 18-bit data access. Therefore, even though only 16 bits are needed for each word, each word will still occupy 18 bits of hardware space. Therefore it is reasonable to assume that each line buffer will occupy $\lceil \frac{N_{max}}{40} \rceil$ and since there are 8 such groups so estimated BRAM will be given by,

$$\text{BRAM}_{\text{Maxpool}} = \left\lceil \frac{N_{max}}{40} \right\rceil \times 8$$

In this design, Maxpool IP uses 8 instances of BRAM as N_{max} is set to 32 each occupying 832 words as shown in figure 6.3. Also, the IP DSP utilization is independent of tunable parameters because the hardware resource required for comparison does not require any multiplication. Only one DSP core is inferred which is utilized in the control path of the design for checking when the line buffers are filled with required data for making comparisons inside the 2×2 window for selecting the maximum value.

	$\mathbf{P_{pool} = 1}$	$\mathbf{P_{pool} = 2}$	$\mathbf{P_{pool} = 3}$	$\mathbf{P_{pool} = 4}$
DSP48E	1 (∼0 %)	1 (∼0 %)	1 (∼0 %)	1 (∼0 %)
FF	1500 (1 %)	1629 (1 %)	1664 (1 %)	1635 (1 %)
LUT	2503 (4 %)	2708 (5 %)	2774 (5 %)	2776 (5 %)
BRAM	8 (2 %)	8 (2 %)	8 (2 %)	8 (2 %)
Initiation interval (II_yolo)	4	2	2	2
Latency (milliseconds)	3.462	1.731	1.732	1.731

Table 6.3: Latency and resource utilization for various values of p_maxpool

For max pooling, the padding value should be set to negative infinity which is -FP_MAX in fixed-point implementation instead of zero. Also, unlike convolutional layers where the input is padded with a ring of zeros, padding for 2x2 max pooling occurs when the height or width of the input is an odd number. The reason for padding in max pooling is due to the use of a 2x2 filter with a stride of 2, which moves over the input in steps of 2 pixels. When the input has an odd height or width, the filter may leave one pixel unprocessed after it has covered the entire input. Padding is then necessary to ensure that this pixel is considered in the pooling operation.

The window max-pool function used in the design is again a tunable parameter whose instances are controlled by the pragma HLS_ALLOCATION. The limit is set to 2 in this design even though 4 channels can be processed in parallel. This is done to minimize resource utilization in case of resource-constrained devices and decreasing the limit does not change the initiation interval of the pipeline and the latency of the IP remains the same. A higher value of limit increases the throughput but causes more resource utilization.

```
for i = 0; i < hh; i++ do //output row
    for j = 0; j < stride_row; j++ do
        for k = 0; k < hw; k++ do  //output column
            for l = 0;l < col_stride ;l++ do
                for m =0; m< Nin; m++ do
                    #pragma HLS PIPELINE II=IImax
                    #pragma HLS ALLocation window Maxpool limit = 2 function
                    line buffer;
                    sliding window function;
                    window Maxpool;
                    write output;
        end
    end
end
```

Listing 6.2: Algorithm of Maxpool IP

6.4.1 Latency estimate of Maxpool Layer

Maxpool Layer latency in clock cycles is given by the product of trip count and Initiation interval of the pipeline. Trip-count depends on h_h, h_w, *spool*, and N_{in}. Due to bandwidth constraints, only 4 channels can be processed in parallel. Therefore the latency is given by,

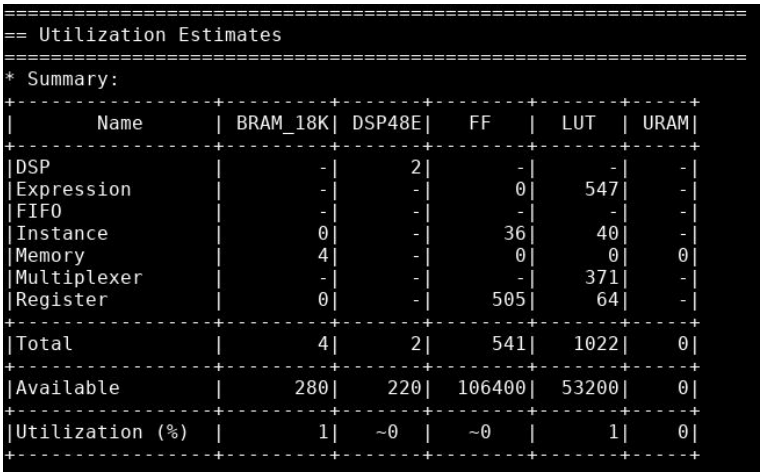
$$\boxed{\text{Latency}_{\max} = \left\lceil \frac{h_h}{2} \right\rceil \times 2 \left\lceil \frac{h_w}{2} \right\rceil \times 2 \times \text{spool} \times \text{spool} \times \left\lceil \frac{N_{in}}{4} \right\rceil \times II_{max}} \tag{6.12}$$

6.5 Upsample Layer IP block

The upsample layer does the opposite of the pooling layer. It takes inputs that correspond to the top-left corner of a 2x2 window and stores them in a line buffer before filling the rest of the window. Unlike the pooling layer, the upsample layer does not pad the inputs with zeros but instead repeats the input values.

Due to the DMA constraint of processing 4 channels parallelly, firstly all 4 channels are processed parallelly using a tunable design parameter $P_{upsample}$. The $P_{upsample}$ can take values between 1 to 4. The $P_{upsample}$ is set to 4 initially. The resource utilization report for this IP is given in table 6.11 which shows the resource utilization in terms of LUT and Flip-flops is less than 1%. The design achieves the initiation interval of 1 at $P_{upsample} = 4$. The latency is expected to be minimum at this value as 4 channels are processed. As resource utilization is already less than 1% and latency is minimum, therefore $P_{upsample} = 4$ is set in this design.

The line buffer which will utilize BRAM's has a size of 1×13. The height is fixed as 1 as the line buffer in the upsample layer holds only the most recent row and can be treated as a line buffer with only one row. The width is fixed at 13 as there is only one upsample layer and it has a width dimension of 13. BRAM utilization of the again depends on the



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      2|       -|      -|    -|
|Expression       |        -|      -|       0|    547|    -|
|FIFO             |        -|      -|       -|      -|    -|
|Instance         |        0|      -|      36|     40|    -|
|Memory           |        4|      -|       0|      0|    0|
|Multiplexer      |        -|      -|       -|    371|    -|
|Register         |        0|      -|     505|     64|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        4|      2|     541|   1022|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |        1|   ~0  |   ~0   |      1|    0|
+-----------------+---------+-------+--------+-------+-----+
```

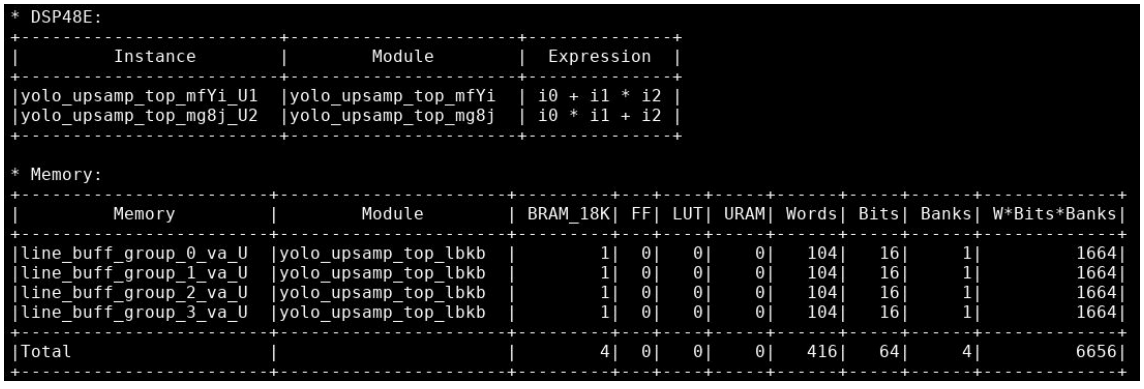
Figure 6.11: Resource utilization for upsample layer

size of the words stored in the BRAM. As a maximum of 32 channels can be processed in a single run, therefore line buffers will store $1 \times 13 \times N_{max}$. Due to DMA constraints, only 4 channels can be processed in parallel. Therefore there are 4 groups of line buffers that can store $13 \times \lceil \frac{N_{max}}{4} \rceil$ words. As BRAM18k can store only 1k words of size 18 bits, therefore it is reasonable to estimate that BRAM utilized by upsample layer will be given by

$$\text{BRAM}_{\text{upsample}} = \left\lceil \frac{N_{max}}{316} \right\rceil \times 4$$

In this design, N_{max} is set to 32 which leads to BRAM utilization to 4, and each BRAM will occupy $13 \times \lceil \frac{32}{4} \rceil = 104$ as shown in figure 6.12. The design utilizes 2 DSP cores for implementing control logic.

The upsample layer in YOLOv4-tiny only occurs once, so there are specific values for its parameters that are predetermined and cannot be adjusted through control ports. Specifically, the height g_h and width g_w of the layer inputs are both 13 and the stride of the sliding window $S_{upsample}$ is 2.



```
* DSP48E:
+-------------------------+----------------------+--------------+
|         Instance        |        Module        |  Expression  |
+-------------------------+----------------------+--------------+
|yolo_upsamp_top_mfYi_U1  |yolo_upsamp_top_mfYi  | i0 + i1 * i2 |
|yolo_upsamp_top_mg8j_U2  |yolo_upsamp_top_mg8j  | i0 * i1 + i2 |
+-------------------------+----------------------+--------------+

* Memory:
+-----------------------+----------------------+---------+---+----+-----+------+-----+------+-------------+
|         Memory        |        Module        | BRAM_18K| FF| LUT| URAM| Words| Bits| Banks| W*Bits*Banks|
+-----------------------+----------------------+---------+---+----+-----+------+-----+------+-------------+
|line_buff_group_0_va_U  |yolo_upsamp_top_lbkb  |        1|  0|   0|    0|   104|   16|     1|         1664|
|line_buff_group_1_va_U  |yolo_upsamp_top_lbkb  |        1|  0|   0|    0|   104|   16|     1|         1664|
|line_buff_group_2_va_U  |yolo_upsamp_top_lbkb  |        1|  0|   0|    0|   104|   16|     1|         1664|
|line_buff_group_3_va_U  |yolo_upsamp_top_lbkb  |        1|  0|   0|    0|   104|   16|     1|         1664|
+-----------------------+----------------------+---------+---+----+-----+------+-----+------+-------------+
|Total                   |                      |        4|  0|   0|    0|   416|   64|     4|         6656|
+-----------------------+----------------------+---------+---+----+-----+------+-----+------+-------------+
```

Figure 6.12: DSP and line buffer utilization for upsample layer

6.5.1 Latency estimation of upsample layer

Upsample Layer latency in clock cycles is given by the product of trip count and Initiation interval of the pipeline. Trip-count depends on g_h, g_w, *spool*, and N_{out}. Due to bandwidth constraints, only 4 channels can be processed in parallel. Therefore the latency is given by,

$$\boxed{\text{Latency}_{\text{upsample}} = g_h \times g_w \times \text{S}_{\text{upsample}} \times \text{S}_{\text{upsample}} \times \left\lceil \frac{N_{out}}{4} \right\rceil \times II_{upsample}} \tag{6.13}$$

6.6 Yolo Layer

The Yolo layer in this design is specifically tailored for YOLOv4-tiny. For a Yolo layer, it takes in an input tensor of size $g_h \times g_w \times N_{in}$ and divides it into $g_h \times g_w$ grids. The number of input channels N_{in} by equation 6.14 where B represents the maximum number of objects that can be detected in one grid, and C is the number of object classes. For YOLOv4-tiny, B is set to 3 and there are 80 object classes in the COCO dataset.

$$\boxed{N_{in} = (4 + 1 + C) \times B} \tag{6.14}$$

The number of input channels N_{in} for the YOLO layer in YOLOv4-tiny is fixed at 255 and is divided into 3 groups. Each group has 85 channels, out of which 4 channels contain information about the bounding boxes, 1 channel is for objectness score, and the remaining 80 channels represent the class scores for individual objects. The YOLO layer applies a sigmoid activation function to all channels except for those corresponding to the width and height of bounding boxes.

The task at hand is to determine which channels require the sigmoid function and which do not. To address this, a table is used, where each bit indicates whether a particular channel should undergo the transformation or not as shown in figure 6.13. Each entry in the table corresponds to the activation associated with 32 channels with 8 such entries, resulting in a total of 256 channels of the YOLO layer. The host provides this table through a straightforward AXI4-Lite connection since only 255 bits need to be transferred. The accuracy of the sigmoid function is crucial in this design, especially because the YOLO layers produce the final results of the network. The sigmoid activation involves a division and an exponential function which can be efficiently implemented on hardware using a fixed-point version of the exponential function provided by Xilinx.

```
u32 activate_en[8]={0xffffffff3,0xffffffff,0xfe7fffff,0xffffffff,0xffffffff,0xffffcfff,0xffffffff,0x7fffffff};
```

Figure 6.13: Yolo-layer channel activation using table

For this block, P_{yolo} is a tunable parameter that controls the hardware for implementing the sigmoid activation function across the channels. This parameter is controlled by pragma

HLS_ALLOCATION in this design. This parameter signifies that a separate instance of the sigmoid activation function will be used for processing. Due to DMA constraints, only 4 channels can be processed in parallel. Therefore this design parameter can be set between 1 and 4. For instance, if $P_{yolo} = 4$, then all 4 channels will have separate instances of the sigmoid activation function. This in turn means all 4 channels are processed in parallel in a single clock cycle. This results in an initiation interval of 1 clock cycle.

Figure 6.4 shows the change in latency and resource utilization with a change in the tunable design parameter P_{yolo}. As there are no line buffers utilized in this block, the block RAM utilization for this IP is 0. The resource utilization such as the number of DSP blocks inferred, Flip-Flops, and LUT utilized depends on the tunable parameter P_{yolo}. As explained earlier, $P_{yolo} = 4$, all 4 channels will have separate instances of the sigmoid activation function and for processing each channel a minimum of 2 DSP cores are needed for fixed-point exponential computation inside the sigmoid activation function. This makes the DSP count 8 in $P_{yolo} = 4$. As the initiation cycle becomes 1, the latency is expected to be the least. But this decrease in latency comes at the price of increased hardware resources as shown in the table. Resource utilization becomes maximum when $P_{yolo} = 4$ as compared to any other value. In the case of $P_{yolo} = 2$, there will be 2 separate instances of the sigmoid activation function available for processing 4 channels. Therefore, the Yolo design pipeline will add extra registers to increase pipeline depth. As a result, 2 clock cycles will now be required to process 3 channels. This also explains why the initiation interval is 2. DSP utilization in this case is 4, which can be explained similarly to what was explained earlier. Please note that $P_{yolo} = 3$ is similar to $P_{yolo} = 2$ as dividing 3 separate hardware resources among 4 channels will require $\frac{4}{3}$ clock cycles, a clock cycle for processing 3 channels parallelly and then a clock cycle for processing 4^{th} channel. As clock cycles cannot be in fractions, HLS tools take the $\lceil \frac{4}{3} \rceil = 2$. The table shows the latency for a single IP call. There are 2 Yolo layers in Yolo4-tiny and each layer has 255 input and output channels. A maximum of 32 channels can be processed in one call. This means a single IP call means the first 32 input channels will be used to process the partial output of the first 32 output channels. After applying the concept of input channel folding, as discussed in chapter 7, the IP will be called 64 times. So the latency numbers will be multiplied by 64 to get the final latency. The difference in latency between $P_{yolo} = 1$ and $P_{yolo} = 4$ is $(54.40 - 31.84) \times 64 = 2.59$ ms is very small compared to the resources utilized for these values of P_{yolo}. Therefore in this design, P_{yolo} is set to 1.

	$\mathbf{P_{yolo} = 1}$	$\mathbf{P_{yolo} = 2}$	$\mathbf{P_{yolo} = 3}$	$\mathbf{P_{yolo} = 4}$
DSP48E	1 (∼0 %)	4 (1 %)	4 (1 %)	8 (3 %)
FF	3614 (3 %)	5867 (5 %)	5867 (5 %)	10562 (9 %)
LUT	2900 (5 %)	4181 (7%)	4181 (7%)	6639 (12 %)
BRAM	0	0	0	0
Initiation interval (II_yolo)	4	2	2	1
Latency (microseconds)	54.40	27.36	27.36	13.84

Table 6.4: Latency and resource utilization for various values of P_{yolo}

6.6.1 Latency estimation of Yolo IP block

Yolo layer IP block latency will be equal to the product of trip-count and initiation interval (II_{yolo}) of the pipeline. The trip count depends on the input topological dimension of the input such as width (g_w), height (g_h), and number of desired output channels (N_{out}). Again, Due to DMA constraints, only 4 channels can be processed in parallel. Therefore the latency of the Yolo layer is given as,

$$\boxed{\text{Latency}_{\text{yolo}} = g_h \times g_w \times \left\lceil \frac{N_{out}}{4} \right\rceil \times II_{yolo}} \tag{6.15}$$

Chapter 7

System Design

This chapter is dedicated to integrating individual IP blocks into a complete system. It begins with a system-level description in section 7.1 which is followed by a discussion on how to modify the YOLOv4-tiny network to make it suitable for FPGA acceleration. Then, it explains the block-level design and software drivers.

7.1 System Overview

The system architecture, as shown in figure 7.1, utilizes several IPs that function as a unified accelerator for YOLOv4-tiny. The system architecture has been followed as given in the original literature but differs in hardware accelerator implementation. The design choices made for different IP blocks and it's implementation is described in chapter 6. To facilitate the transfer of large volumes of data, the off-chip DDR is connected to the CNN accelerator via AXI4-streams. DMA0 has two data ports, one for reading and the other for writing, each with a width of 64 bits. DMA1 only moves data from main memory to the slave and is used for accumulation within the accelerator.

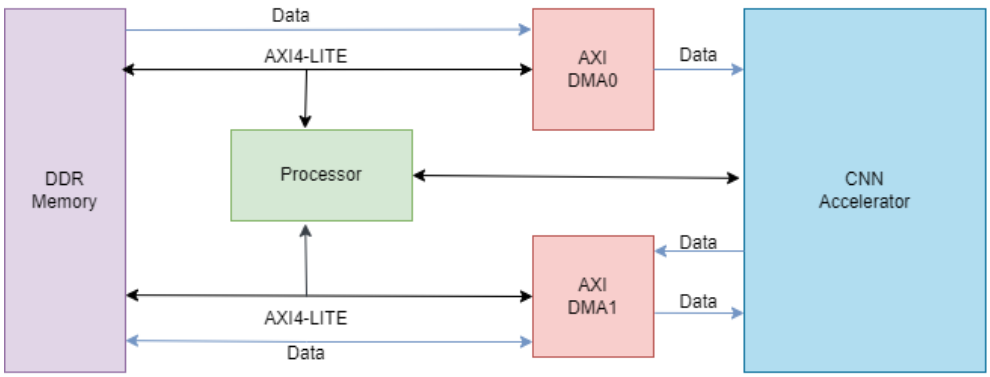


Figure 7.1: System Design [1]

The processor is in charge of managing the system using the bundled AXI4-Lite interface.

The connection between the processor system (PS) and the accelerator is also utilized for transmitting typology parameters. The figure doesn't show all the modules, such as Processor System Reset and AXI Interconnect.

7.2 Network Shaping

Even though this design allows for dynamic configuration, the network has to be reshaped to take into account the maximum channels that can be processed by the hardware IP. The number of bits supported by the AXI interface is dependent on the design parameter N_{max}. The maximum value of N_{max} as per the yolov4-tiny architecture is 512. In order to process 512 output channels with the 16-bit quantization, the AXI interface has to support 8192 bits which is beyond the maximum supported bandwidth of 1024 bits. Moreover, for the best case of 1024 bits, the number of channels that can be processed in a single execution can be 64 channels. But this will require 234 DSP blocks which is beyond what is available on Zedboard. Therefore, the topology must be transformed to fit the available hardware. These transformations will change the network structure but will not affect the final outputs.

Assuming that a convolutional layer contains N_{in} input channels, the layer will consist of N_{out} convolution filters. These filters operate according to equation 7.1 to transform the inputs into N_{out} output channels. In this context, f_i represents the i_{th} input channel, while g_j represents the j_{th} output channel.

$$g_j = \sum_{i=1}^{N_{\text{in}}} f_i * w_{i,j} + b_j, \quad \text{with} \quad j \in [1, N_{\text{out}}] \tag{7.1}$$

7.2.1 Channel Folding

The maximum number of channels (N_{max}) that can be used in a convolutional layer is limited by the available resources. This means that N_{max} may be smaller than the desired number of input channels (N_{in}) or output channels (N_{out}). To overcome this limitation, the original layer can be split into sub-layers with fewer channels that can fit into the available IPs. This technique is called channel folding. When a convolution layer is folded, all other layers are also adjusted to avoid re-interleaving the channels.

- ***input channel folding***

  For a convolutional layer with N_{in} input channels, if N_{out} is smaller than N_{max}, there will be $\lceil \frac{N_{in}}{N_{max}} \rceil$ sub-layers in which input channels will be divided. $\lceil \frac{N_{in}}{N_{max}} \rceil$ is also defined as input channel folding factor F_{in}. If N_{in} is divisible by N_{max}, each sub-layer will contain N_{max} input channels and N_{out} output channels [1]. The outputs of these sub-layers are $g_{j;1}, g_{j;2}, ...; g_{j;Fin}$. The accumulation and activation module is applied to these results to obtain the final outputs g_j. This process can be mathematically

explained by splitting equation 7.1

$$g_j = \sum_{i=1}^{N_{\max}} f_i * w_{i,j} + \sum_{i=N_{\max}+1}^{2N_{\max}} f_i * w_{i,j} + \ldots + \sum_{i=(F_{\text{in}}-1)N_{\max}+1}^{N_{\text{in}}} f_i * w_{i,j} + b_j, \quad \text{with } j \in [1, N_{\text{out}}] \tag{7.2}$$

To put it differently, input channel folding divides the accumulation chain into smaller groups. If N_{in} is larger than the available resources, it needs to be divided into F_{in} sub-layers with N_{max} input channels each. Theoretically, the total latency of these sub-layers should be equal to that of the original layer. However, in practice, there is an additional cost associated with launching the IP multiple times, which affects the overall latency.

- ***output channel folding***

  In contrast, output channel folding happens when N_{out} is larger than N_{max}. Unlike input folding which will not affect total latency much, the output channels folding factor will contribute to the latency linearly [1].

  When output channels are folded, all inputs have to be sent again, which increases the overall latency of one layer by a factor of F_{out} (or F_{in} if input channel folding is applied). This is because the data is a stream and only part of the inputs are buffered locally. Additionally, as mentioned before, starting IPs multiple times is an extra cost. Figure 7.2 can be used as an example to understand the concept of channel folding wherein the original layer group with 64 input channels and 64 output channels has been partitioned such that a maximum of 32 input and output channels will be processed. In the first step, the first 32 input channels ($Conv\_Sub1.1$) are used to generate the partial output of the first 32 output channels. Then in the second execution, the remaining 32 channels ($Conv\_Sub1.2$) are processed, and the partial sum of this execution is added to the previous one using (ACC_Sub1) to generate outputs of the first 32 channels. A similar step is followed for the next 32 output channels which is shown in the figure.

7.2.2 Channel Padding and Kernel size padding

The DMA port can transfer four channels at the same time, so it's better if the number of channels in a layer is divisible by 4. This is usually the case, except for RGB inputs which have only three channels i.e. First Layer and YOLO layers with 255 channels. Only 3 out of the 38 layers will be affected by this, so an extra channel is added in this design to these layers to make them divisible by 4. In the YOLOv4-tiny neural network, the convolution kernel size is limited to either 3x3 or 1x1. Since the convolution is already "unrolled" as discussed in chapter 6, it is difficult to share resources between the two different kernel sizes. In other words, it is not possible to reconfigure a 3x3 kernel into nine 1x1 kernels using the same

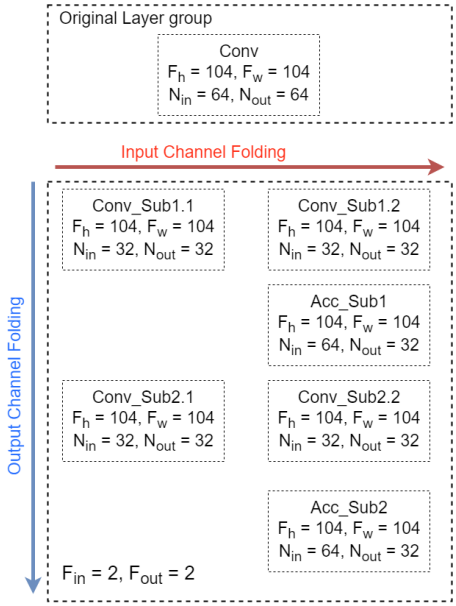


Figure 7.2: Concept of Channel folding with example

resources. To incorporate 1x1 convolution into the YOLOv4-tiny network, an alternative method is used where 1x1 convolution is padded into a 3x3 convolution. To achieve this, the kernel size is transformed and the input remains unchanged, but the 1x1 weights are surrounded by zeros to create a 3x3 window. Although the idle zeros in the weights contribute to the computation, they do not contribute to the generation of meaningful outputs or affect the final results of the convolution operation. However, this comes at a price of extra latency as these zeros will contribute to additional latency.

7.3 Hardware Accelerator Block setup

Once the neural network has been transformed, the IPs designed in previous chapter can be classified into three types based on their functions. The first type of IP is responsible for performing depth separable convolutions. It fetches weights and input data through DMA and then feeds the data into an accumulation and activation module. The second type of IP is involved in the accumulation process, which takes place when input channels are folded prior to biasing and activation. The third type of IP handles the accumulated and activated outputs. These outputs are processed further by modules such as max pooling, yolo, or up-sample, depending on the topology of the network.

In this thesis, two setups are designed for accelerating yolov4-tiny shown in figure 7.3. The first setup (a) accelerates only convolutional operations and accumulation layer operations. The rest of the layer operations including bounding box detections and Non-max suppression are done in PS inside the Arm-A9 cortex. The second setup (b) accelerates all the layers but keeps the routing layer, Bounding box detection, and Non-max suppression inside the PS only. The detailed diagram of the second setup is shown in figure 7.4

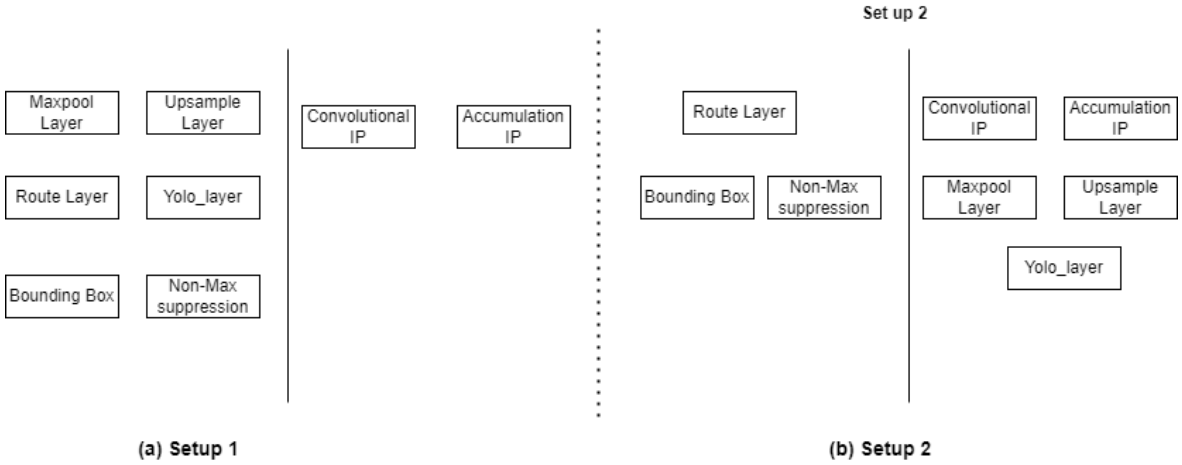


Figure 7.3: Setup for hardware acceleration

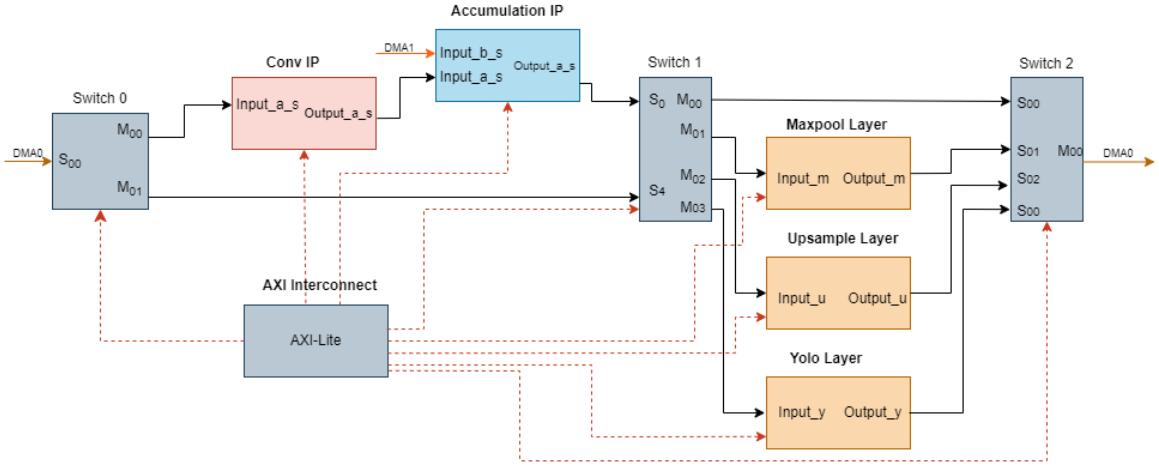


Figure 7.4: Hardware accelerator block diagram [12]

As a result, the Vivado block design of the system includes three processing stages on the programmable logic (PL). These stages are connected via AXI4-stream and switch IPs, which enable configurable routing between masters and slaves. In addition, the programmable logic (PL) and processing system (PS) communicate with each other through two 64-bit DMA ports and an AXI4-Lite interface. Switches 0 and 1 have the ability to control the direction of input data flow for DMA0. The data can either go into the convolutional IP or bypass it and go directly to the third stage. While it may seem efficient to combine convolutional layers with other types of layers, it is not always the best approach. This is especially true when the outputs of a convolutional layer are needed elsewhere in the network, requiring the results to be captured by the off-chip memory. In such cases, even if a pooling layer follows the convolutional layer, the two layers must be executed separately. Therefore, there may be situations where the convolutional IP is bypassed. Switch 1 and Switch 2 are responsible for determining the type of procedure that needs to be performed on the accumulation outputs, such as max pooling, yolo, upsample, or no action at all. If there is no additional processing,

it could indicate that the subsequent layer involves convolution or that the accumulation of channel folding is still ongoing.

7.4 Processing System (PS) Design

7.4.1 Software Driver for ARM A9 Cortex Processor

The processing system's drivers shown in listing 7.1 are responsible for performing three tasks: initializing all peripherals, categorizing layer groups based on their typology, and passing on all groups to obtain results. The first two tasks are considered part of the system initialization and should only be carried out once. To achieve the detection of multiple images or processing of a video stream, the forwarding stage of the processing system can be divided into five distinct steps. These steps involve setting IP parameters, creating routes through switches, initiating IPs, transmitting input stream data, and retrieving outputs. However, in order to accomplish this task, the forwarding stage must be executed repeatedly. Furthermore, the core of the forwarding stage consists of a nested loop. The outer loop is responsible for iterating over the output channel folding, while the inner loop covers the input channel folding. To handle input channel folding, buffers are required on DDR to store both the accumulation inputs and outputs. For accumulation, it is not a good idea to copy data from the output buffer to the 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 the input buffer and output buffer have the same size. This further supports the idea that the maximal number of input channels.

```
1
2  for i = 0; i <Fout; i++ do
3      for j = 0; j <Fin; j++ do
4          Set IPs by setting topological Parameters;
5          Start IPs;
6          if Layer group contains Conv then
7              send weights;
8              if j == Fin - 1 then
9                  send biases;
10             end
11         end
12         DMA transfer;
13         wait until all IPs are done;
14         swap pointers of accumulation buffers;
15     end
16 end
```

Listing 7.1: Software Driver Code

7.4.2 Weights rearrangement for channel folding

At present, there is no specific hardware available for a 1x1 convolution kernel. As a result, the weights of those layers have to be padded to 3x3 convolutions. Additionally, the weights are transferred in the form of a kernel window unit. However, the size of a 3x3 kernel is not divisible by 4, which creates the need for encoding and decoding circuits to fully utilize the 64-bit bandwidth. This results in an average of 2.25 cycles per window. Alternatively, the kernel size can be padded to 12, which takes 3 full cycles to transfer each 3x3 window.

To enable channel folding, it is necessary to rearrange the weights of the convolutional layer beforehand. The weights are initially stored in a 3-dimensional array, with the three dimensions representing kernel windows, input channels, and output channels. However, during one DMA transfer, only a maximum of N_{max} out of the N_{in} input channels are accessed, and they are not stored in a contiguous space. This means that the input channels need to be reorganized before runtime, as illustrated in Figure 7.5.

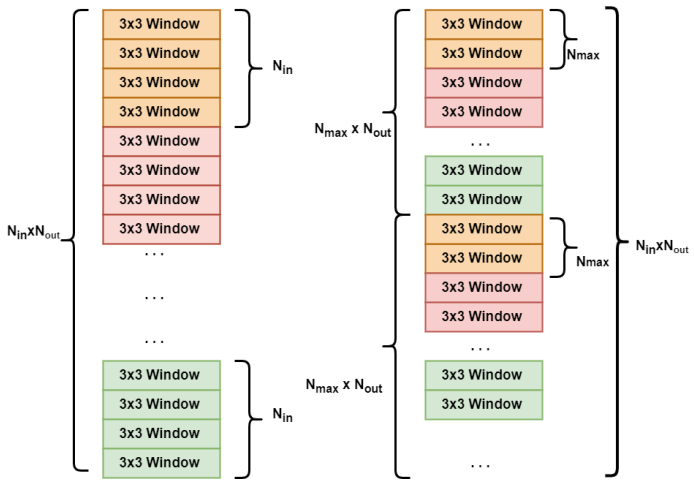


Figure 7.5: Weights rearrangement for channel folding

7.4.3 Memory Access

Calculating the total number of memory transactions for a layer group is an important task. The variables f_h, f_w, and N_{in} represent the input height, input width, and a number of input channels for the layer group, while h_h, h_w, and N_{out} represent the output height, output width, and a number of output channels for the same group.

- ***Total input size*** $(size_{in})$: $f_h \times f_w \times \lceil \frac{N_{in}}{4} \rceil \times 4$
- ***Total output Size***$(size_{out})$: $h_h \times h_w \times \lceil \frac{N_{in}}{4} \rceil \times 4$
- ***Total Accumulator Size***$(size_{acc})$: $h_h \times h_w \times \lceil \frac{N_{in}}{4} \rceil \times 4$
- ***Total Weight Size***$(size_{weight})$: $\lceil \frac{K_h \times K_w}{4} \rceil \times 4 \times N_{in} \times N_{out}$

- ***Total Bias Size***$(size_{bias})$: $\lceil \frac{N_{out}}{4} \rceil \times 4$

This memory access equation is later used for the estimation of the total latency caused by each layer group. This estimation gives the fraction of the total latency caused by executing the software driver code in the PS. Additionally, these models will be used for carrying out design space exploration which is discussed in section 7.5

7.4.4 Input Image transformation

One aspect that has not been discussed yet is the system's image input. The dimensions of the network input are fixed at 416 pixels for both height and width. If the input image has a different size, it must be first resized and then letterboxed. Afterward, the pixel values are normalized between 0 and 1, and channels interleaving and padding are applied. Padding is a method utilized to include additional pixels surrounding the image's edges, making sure that the filters can be used for all pixels. Usually, this is carried out by appending zeros around the edges of the image. This pre-processing step is carried out before run-time, as the primary focus is on the YOLO network itself. The image is required to be available in off-chip memory before network detection as a header file for a single image at a time.

7.4.5 Route Layer Implementation

The route layer is implemented in software rather than FPGA hardware, as it solely involves copying and moving data. Its main purpose is to redirect the flow of data within the network, which is essential for detecting objects at various scales. If the route layer only has one input layer, it is implemented by merely passing a pointer. However, when multiple input layers are present, the data is copied into a contiguous memory space, which effectively concatenates the data of input layers on the channel dimension.

7.5 Design Space exploration

The latency and resource utilization estimation for various IP blocks as discussed in chapter 6 provides estimates on the usage of resources like latency, BRAM, and DSP utilization based on the design parameter vector. This design parameter vector maps to specific resources and latency, which together form the design space. By exploring the design space, we can understand the configurability of the architecture. Design Space Exploration (DSE) provides an efficient way to find optimal design points, which can achieve the lowest latency while using minimal resources.

Figure 7.6 and 7.7 illustrates the variation in latency with DSP and BRAM unit for the yolov4-tiny model. When designing for a specific platform with a limited set of resources, in this case, Zedboard, it is important to determine the maximal resources available for that platform. DSE analysis is carried out using a Python script that takes into account the latency estimates of each IP block and applies channel folding in the case when the number of input and output channels increases beyond. 32. The Python model takes into account

the range of tunable parameters for exhaustive DSE. The range of N_{max} is in multiples of 4 between 4 to 64 channels due to DMA constraints. The convolutional IP block tunable parameter P_{mem} which controls the factor by which local memory is partitioned is set in the range from 1 to 32 in multiples of 2. The range of another tunable parameter such as P_{acc},P_{pool},P_{yolo} is again set between 1 to 4 due to DMA and 16-bit fixed-point quantization constraint. The envelope of the graphs shows the range of achievable design points for the given set of resources and allows for the selection of the most optimal design point that meets the requirements of the design.

The graphs also reveal that optimal points have an inverse relationship, meaning that

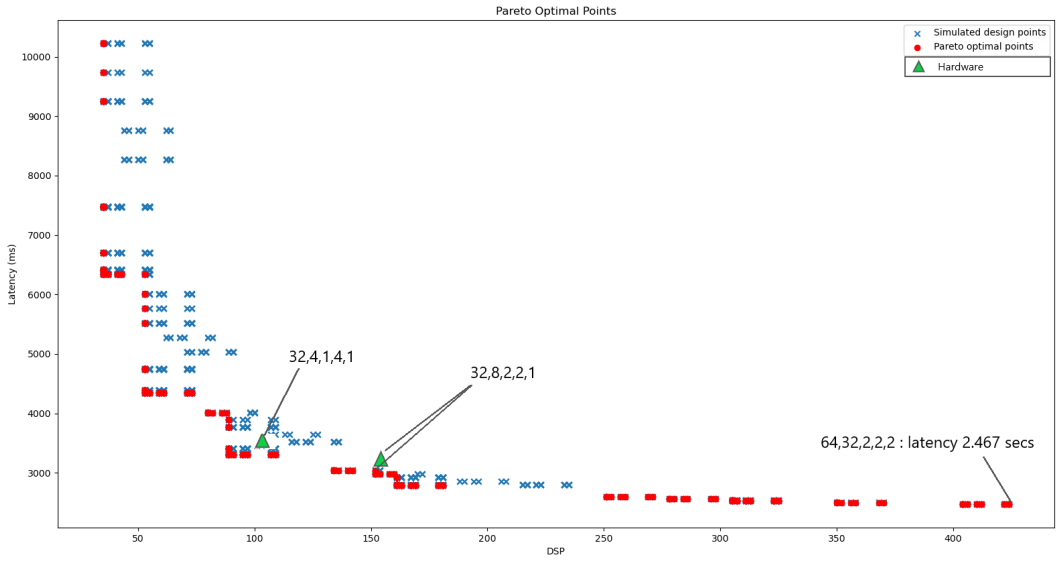


Figure 7.6: Latency vs DSP utilization

as the available resources increase, the slope of the fitting function decreases. This indicates that simply doubling resources will not result in the same level of speed improvement. Additionally, the simulation shows that even with using a target platform with more DSP blocks that can process more channels in parallel, for instance, 64 channels, the estimated maximum performance the current design achieves is 2.467 sec. This means that other factors such as memory bandwidth and software costs can also affect latency, and these are not impacted by improvements in the computation capability of hardware IPs. The distribution of design points for BRAM is more sparse as compared to DSP, indicating that it is more likely to make poor decisions resulting in under-utilization of BRAM. To increase parallelism in HLS, arrays are partitioned to create additional memory ports. However, when the size of the array is small, each 18Kb BRAM cannot be fully utilized, leading to a waste of memory space.
In order to check how the estimated latency and resource utilization deviates from the actual values, 2 design points are tested on the board. The coverage of the test is low because of the long execution time of re-packaging the IP, Vidado synthesis, placement and routing of

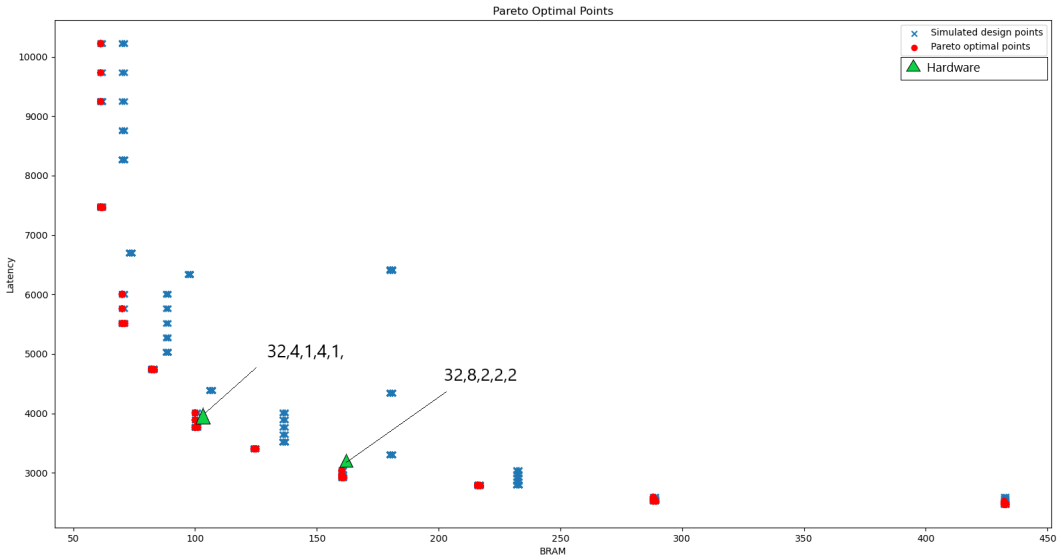


Figure 7.7: Latency vs BRAM utilization

the design, and bitstream generation. Additionally target clock period in the HLS has to be modified again until the design meets the time constraints after placement and routing. The two test points are represented in black dots in Figure 7.6 and 7.7. The estimated latency for the design point (32,8,2,2,1) is 3.020 secs and the latency measured on hardware is 3.216 sec. The estimated latency of the other design point (32,4,1,4,1) is 3.410 sec and the measured latency is 3.623 sec. The deviation in latency within both cases is less than 7%. Additionally, the estimated resource utilization in the first design point (32,8,2,2,1) has DSP cores 161 and a BRAM utilization of 155 blocks. The differs by 5% as compared to the measured resource utilization of 161 DSP cores and 148 BRAM18K blocks.

Chapter 8

Results

In this chapter, a performance comparison is made with respect to this implementation (PS vs PL) as well as earlier implementation for yolov4-architecture on resource-constrained FPGA platforms like Zedboard. As the discussed architecture is more scalable and can be implemented on other Xilinx-based target platforms, the Zedboard here is chosen as an indicative platform. Unfortunately, implementation on other platforms has not been tested due to lack of time. The evaluation of this work primarily focuses on six key aspects, which are layer-wise Performance improvement, target network platforms, resource utilization, speed, and power consumption.

The chapter starts with introducing different target platforms on which the YOLOv4-tiny model was tested in section 8.1. Section 8.2 discusses the unified architecture tunable parameters and their associated values for different IP blocks. Section 8.3 discusses layer-wise performance improvement with respect to the PS implementation of this work and with other literature work on YOLOv4-tiny with the same target platform. Section 8.4,8.5, 8.6, and 8.7 discusses the resource utilization breakdown, target platforms, latency, and power consumption respectively of the current setup with respect to the previous works. Finally, section 8.8 discusses the comparison of the performance of the FPGA implementation with similar implementations on other platforms like CPU and PS of the ZedBoard to demonstrate the advantage of using FPGA.

8.1 Specifications of the target platform

DNNs are typically executed on commonly used hardware platforms, such as CPUs and hardware platforms like the Arm cortex A9 processor of the ZedBoard. Therefore, it is crucial to use these platforms as a benchmark to gain a comprehensive understanding of how the accelerator impacts performance.

8.1.1 Intel Core i5-8250U CPU

Intel corei5-8250U is used as the CPU platform for running the YOLOv4-tiny simulation model. The specification of the host CPU is given in table 8.1. The Darknet framework is

used for obtaining the latency results of each layer. Darknet is optimized for running deep neural networks (DNNs) on CPUs and GPUs, as well as on specialized accelerators such as FPGAs and ASICs. It is designed to be flexible and efficient, allowing for high performance and accuracy in a variety of applications.

Specification	Cores	Base Frequency	Max Turbo Frequency	Cache	Max Memory Bandwidth
Value	4	1.6 GHz	3.4 GHz	6 MB Intel smart cache	41.8 GB/s

Table 8.1: CPU specifications

8.1.2 ZedBoard

The software application and hardware accelerator developed in this study will be executed on the ZedBoard, which incorporates the Zynq7020 all-programmable system-on-chip (SoC). Table 8.2 provides the specifications for the processing system (PS) of the Zynq7020, while Table 8.3 outlines the specifications for the programmable logic (PL).

PS Specification	**Value**
Cores	Dual-core ARM Cortex-A9 MPCore
Max CPU Frequency	667 MHz
Memory	1 GB DDR3 SDRAM (2x 512MB)
Max Memory Frequency	533 MHz
Cache	32 KB L1 instruction cache, 32 KB L1 data cache, 512 KB L2 cache
Max Memory Bandwidth	8.5 GB/s

Table 8.2: Avnet ZedBoard Revision E PS Specifications

Specification	**Value**
Programmable Logic Cells	85,000
LUTs	53,200
FlipFlops	106,400
Block RAM (36 Kb Blocks)	4.9 MB (140 Blocks)
DSP Slices (18x25 MACCs)	220
Max Memory Frequency	250 MHz

Table 8.3: ZedBoard Revision E Programmable Logic Specifications

8.2 Unified accelerator Configurations

The architecture performance and utilization of resources depend upon how the accelerator is configured. Some of the tunable parameters of all IP blocks are shown in grey that can be configured depending on the resources available on the target FPGA.

- The maximum number of out channels and input channels that an IP can process N_{max} is set to 32 due to resource constraints. This is the same for all the IP blocks shown in table 8.4,8.5,8.6,8.7.

- In order to implement channel interleaving, the line buffer size for the convolutional IP, is set to 3×418, as indicated in table 8.4. An additional 2 is added to the buffer width to accommodate padding. For the Maxpool IP, the line buffer size is set to 2×104 based on the typological parameters of the first Maxpool layer as shown in table 8.5. Similarly, for the upsample layer, the line buffer size is set to 1×13 based on the typological parameters of the first upsample layer as shown in table 8.6.
- The block partition factor P_{mem} determines the partitioning of the local memory used for storing weights in the convolutional IP block, as specified in table 8.4. In this design, P_{mem} is set to 8.
- The bias values stored in local memory are partitioned by a factor P_{acc} to optimize the design. The size of the bias memory is set to 32, which aligns with the maximum number of channels that the IP can process. In this case, the bias memory is cyclically partitioned by a factor of 2 for efficient utilization of resources and improved latency as shown in table 8.4.
- The maximum kernel size is set to 3x3 as indicated in table 8.4. 1x1 convolution is also padded to 3x3 in order to simplify the design as explained earlier in chapter 7.
- The stream transmission size is set to 8 in this design as indicated in table 8.4. This choice is made because the AXI interface used in the design can only support 4 input channel data at a time. By setting the transmission size to 8, the design is able to process 32 channels in a single call of the IP.
- The tunable design parameter P_{pool} and P_{yolo} are the factor that denotes the resource sharing of the Maxpool function and logistic function for Maxpool IP and Yolo IP block using HLS ALLOCATION pragma as indicated in table 8.5 and 8.7. These are set to 2 and 1 respectively.

Parameter	Value
Maximum channel Processed N_{max}	32
Memory partition factor P_{mem}	8
Bias partition factor P_{acc}	2
Maximum kernel size	3
Maximum line buffer width	416+2
Stream transmission size	8

Table 8.4: Convolutional and Accumulation IP parameters

Parameter	Value
Kernel dimension	2
line buffer size	2x104
Maximum channel processed N_{max}	32
Maxpool resource allocation P_{pool}	2

Table 8.5: Maxpool IP parameters

Parameter	Value
Maximum channel processed N_{max}	32
line buffer size	1x13
Stride	2

Table 8.6: Upsample layer IP parameter

Parameter	Value
Maximum channel processed	32
Logistic function resource allocation p_{yolo}	1

Table 8.7: Yolo layer IP parameters

8.3 Layer-wise performance comparision

This section discusses the layer-wise performance improvement of the YOLOv4-tiny network architecture. In particular, the performance improvement in terms of latency is discussed. To compare performance, the study [8] was considered because it uses the same object detection algorithm (YOLOv4-tiny) and hardware platform (Zedboard) as the implementation being evaluated, although there are differences in the hardware architecture. However, the comparison only considers the time taken to process the convolutional layer while disregarding the time spent pre-processing input images.

8.3.1 Performance breakdown per convolutional layer: PS only configuration

The performance of each convolutional layer of the YOLOv4-tiny model, when executed on the Arm A9-Cortex processor (i.e. the PS part of the Zedboard), is compared in Table 8.8.

Layer	**Ref [8]** (secs)	**This Work** (secs)	**Speedup Factor**
Convolution Layer 1	3.02	2.99	1.01
Convolution Layer 2	13.57	11.66	1.17
Convolution Layer 3	26.51	22.94	1.15
Convolution Layer 4	6.70	6.22	1.08
Convolution Layer 5	6.67	6.23	1.07
Convolution Layer 6	3.05	2.77	1.10
Convolution Layer 7	26.24	21.93	1.20
Convolution Layer 8	6.55	5.69	1.15
Convolution Layer 9	6.55	5.69	1.15
Convolution Layer 10	2.99	2.55	1.17
Convolution Layer 11	25.48	21.39	1.19
Convolution Layer 12	6.40	5.45	1.17
Convolution Layer 13	6.39	5.45	1.17
Convolution Layer 14	3.00	2.43	1.24
Convolution Layer 15	24.14	21.17	1.14
Convolution Layer 16	1.49	1.19	1.26
Convolution Layer 17	12.09	10.60	1.14
Convolution Layer 18	1.49	1.71	0.87
Convolution Layer 19	0.37	0.30	1.23
Convolution Layer 20	38.21	32.01	1.19
Convolution Layer 21	2.98	2.37	1.26
Total	**223.9 s**	**193.19 s**	**1.15**

Table 8.8: Comparison of layer latency between reference [8] and this Work.: PS only

Table 8.8 compares the two implementations and clearly, this works YOLOv4-tiny simulation model has a better performance w.r.t to the Ref [8] implementation. The speed-up factor is about 1.15s as compared to the Ref [8] implementation indicating that this implementation is about 16% faster than the reference implementation. This speed-up can be explained due to the reason that the Darknet framework is more optimized for performing Yolo algorithm as compared to TensorFlow. The Darknet framework is lightweight and written in C, which

allows for faster computation and lower memory usage. The reason why Darknet has better performance has been discussed in chapter 2.

8.3.2 Performance Breakdown per convolutional Layer: PS+PL Only Configuration

The performance comparison of convolutional layers is shown in Table 8.10 for the PS+PL configuration in comparison to setup 1 of this work. The execution time of only convolutional layers is compared in this table. The PS+PL configuration is found to be 20 times faster than the PS-Only configuration. Table 8.11 provides the execution time of the other layers executed in the PS part. The PS part takes a total of 624.27 ms and the PL part takes 3.03 secs, making a total of 3.654 secs for the entire setup for single image inference. It can also be observed that the execution time of convolutional layers is significantly reduced, such that their execution time is even less than other layers implemented on PS. For instance, the execution time of several convolution layers (1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, and 19) is shown in grey in table 8.10 is less than that of the Maxpool layer number 10 and 12, which takes 192 ms and 96 ms respectively as shown in red color in Table 8.11

Layer	**Ref [8]** (ms)	**PS+PL** (ms)	**Speed up**
Convolution Layer 1	258	114.127	2.26
Convolution Layer 2	2678	258.287	10.38
Convolution Layer 3	3921.85	152.516	25.71
Convolution Layer 4	969.13	44.039	22.02
Convolution Layer 5	969.13	44.120	22.00
Convolution Layer 6	1446.1	152.394	9.49
Convolution Layer 7	3939.8	154.848	25.42
Convolution Layer 8	984	41.441	23.70
Convolution Layer 9	984.9	41.484	23.67
Convolution Layer 10	1443.56	154.812	9.32
Convolution Layer 11	5302.43	201.782	26.27
Convolution Layer 12	1319.61	50.503	26.16
Convolution Layer 13	1319.9	50.426	26.15
Convolution Layer 14	1445.8	201.693	7.16
Convolution Layer 15	13123.2	410.833	31.95
Convolution Layer 16	859.89	205.418	4.19
Convolution Layer 17	6504.55	199.339	32.63
Convolution Layer 18	1486.33	202.402	7.34
Convolution Layer 19	224.37	49.795	4.50
Convolution Layer 20	6140.62	302.571	20.28
Convolution Layer 21	2982.15	289.276	103.19
Total	59599 ms	3020 ms	19.64

Table 8.9: PS+PL Ref ([8]) vs PS+PL (this work) : Setup 1

Even though the other layer contribution is about 17 % of the total time taken by the

Layer	**Heinsius 2021** (ms)	**PS+PL** (ms)	**Speed up**
Convolution Layer 1	258	114.127	2.26
Convolution Layer 2	2678	258.287	10.38
Convolution Layer 3	3921.85	152.516	25.71
Convolution Layer 4	969.13	44.039	22.02
Convolution Layer 5	969.13	44.120	22.00
Convolution Layer 6	1446.1	152.394	9.49
Convolution Layer 7	3939.8	154.848	25.42
Convolution Layer 8	984	41.441	23.70
Convolution Layer 9	984.9	41.484	23.67
Convolution Layer 10	1443.56	154.812	9.32
Convolution Layer 11	5302.43	201.782	26.27
Convolution Layer 12	1319.61	50.503	26.16
Convolution Layer 13	1319.9	50.426	26.15
Convolution Layer 14	1445.8	201.693	7.16
Convolution Layer 15	13123.2	410.833	31.95
Convolution Layer 16	859.89	205.418	4.19
Convolution Layer 17	6504.55	199.339	32.63
Convolution Layer 18	1486.33	202.402	7.34
Convolution Layer 19	224.37	49.795	4.50
Convolution Layer 20	6140.62	302.571	20.28
Convolution Layer 21	2982.15	289.276	103.19
Total	59599 ms	3020 ms	19.64

Table 8.10: PS+PL Ref ([8]) vs PS+PL (this work) : Setup 1

entire PS+PL (3.64ms), An attempt was made to make a unified architecture in which all the other layers were executed on hardware in the given time frame of the project. Keeping that in mind, a second set-up was made which has all the layers accelerated in hardware except the route layer. Please note that bounding box detection and non-max suppression are still being done in PS only as discussed in chapter 7.

8.3.3 Performance Improvement in Setup 2

Setup 2, as described in the Appendix, involves executing all layers in the programmable logic (PL) except for the Route Layer. In this setup, the execution times of the convolutional layers remain the same. However, the execution times of the other layers show significant improvements. For example, according to table 8.12, the execution time of the Maxpool layer decreases by a factor of 8, the Upsample layer by a factor of 4.5, and the Yolo Layers by a factor between 8 to 11. Overall, the total execution time decreases by a factor of 3 for the non-convolutional layers. Therefore the final inference time of the unified accelerator (setup 2) is 3.3 seconds

8.3.4 Performance difference in this work vs Ref [1]

The difference in the performance of Ref [1] and this work can be attributed to the following reasons:

No	Layer	Simulation Time (ms)
4	Route Layer 1	12.121
7	Route Layer 2	24.172
9	Route Layer 3	48.288
10	Maxpool Layer 1	192.997
12	Route Layer 4	5.934
15	Route Layer 5	11.939
17	Route Layer 6	23.628
18	Maxpool Layer 2	96.559
20	Route Layer 7	2.969
23	Route Layer 8	5.982
24	Route Layer 9	11.938
26	Maxpool Layer 3	49.032
31	Yolo Layer 1	23.623
32	Route Layer 10	1.529
34	Upsample Layer 1	18.733
35	Route Layer 11	8.945
39	Yolo Layer 2	93.963
Total Time		624.7 ms

Table 8.11: Execution time of other Layers in PS: setup 1

- Number of convolutional layers in yolov3-tiny is 13 and there are about 21 convolutional layers in YOLOv4-tiny. As convolutional layers are the most computationally intensive elements, more latency in the case of YOLOv4-tiny is expected as the network is deeper in the case of YOLOv4-tiny as compared to yolov3-tiny even though the typological features of each layer in both the algorithms are different.

- The hardware architecture implemented for yolov3-tiny is designed in such a way as to optimize the performance of each layer, not just the convolutional layer. In, yolov3-tiny each convolutional layer is followed by either Maxpool Layer, up-sample layer, or yolo-layer. There are 6 Convolutional-Maxpool layer combinations, 2 Convolutional-Upsample layer combinations, and 2 Convolutional-Yolo layer combinations. Whereas in YOLOv4-tiny, there are no such Convolutional-Maxpool combinations to take advantage of layer combinations. The architecture follows the layer combination transformation technique to compute layer combinations in an efficient manner by reducing the access times to fetch intermediate outputs from the off-chip DRAM [12]. An example is illustrated in Figure 8.1. In the given example, the input folding factor will be 1 and the output folding factor will be equal to 2. This means in a single pass, 32 channel inputs will be used to produce an output of the first 32 output channels which involves both convolutions and accumulations. The output of the accumulation unit is then transferred to Maxpool IP through multiplexing action as mentioned previously in the architectural detail. Maxpool IP process the output for the first 32 output channels using accumulated output from the Accumulator IP. Since the output fold factor is 2, this process is repeated again to get the final output. This simultaneous execution of

Layer Type	Execution Time (ms)
Route Layer 1	12.121
Route Layer 2	24.172
Route Layer 3	48.288
Maxpool Layer 1	23.361
Route Layer 4	5.934
Route Layer 5	11.939
Route Layer 6	23.628
Maxpool Layer 2	12.155
Route Layer 7	2.969
Route Layer 8	5.982
Route Layer 9	11.938
Maxpool Layer 3	6.412
Yolo Layer 1	2.832
Route Layer 10	1.592
Upsample Layer 1	4.039
Route Layer 11	8.945
Yolo Layer 2	11.271
Total	215 ms

Table 8.12: Improved latency for non-convolutional layers (in milliseconds)

2 layers reduces the latency. In YOLOv4-tiny, the execution of 2 layers is happening in a more sequential manner because layer combinations principles cannot be applied as the convolutional layer is not directly followed by the Maxpool layer as shown in Appendix 2.4 So execution has to happen in a sequential manner for both blocks. So in the case of yolov3-tiny, the output channel folding factor will contribute to the latency linearly. When output channels are folded, all inputs have to be sent again. Because data is a stream and only part of inputs is buffered locally. The overall latency of both layers will increase by a $\frac{F_{out}}{F_{in}} = 2$ for both layers' execution. Whereas in the case of YOLOv4-tiny, overall latency will be the sum of individual layer latency i.e. Convolution-route-Maxpool. Firstly, convolutional layer latency increases by a factor of 2 due to the output channel fold and for Maxpool latency increases by a factor of 2. Therefore, in the case of YOLOv4-tiny, the network architecture shape and the Output fold factor of the layer cause more access time to access off-chip DRAM for intermediate results. Please note that cost of launching IP multiple times still does exist in both cases.

8.4 Resource Utilization Breakdown

8.4.1 Resouce Utilization Breakdown: setup 1

The primary calculations carried out by the accelerator involve performing MAC operations, which are assigned to DSP slices. This mapping of MAC operations to DSP slices is a reasonable choice because DSP slices are specialized for such arithmetic operations. The

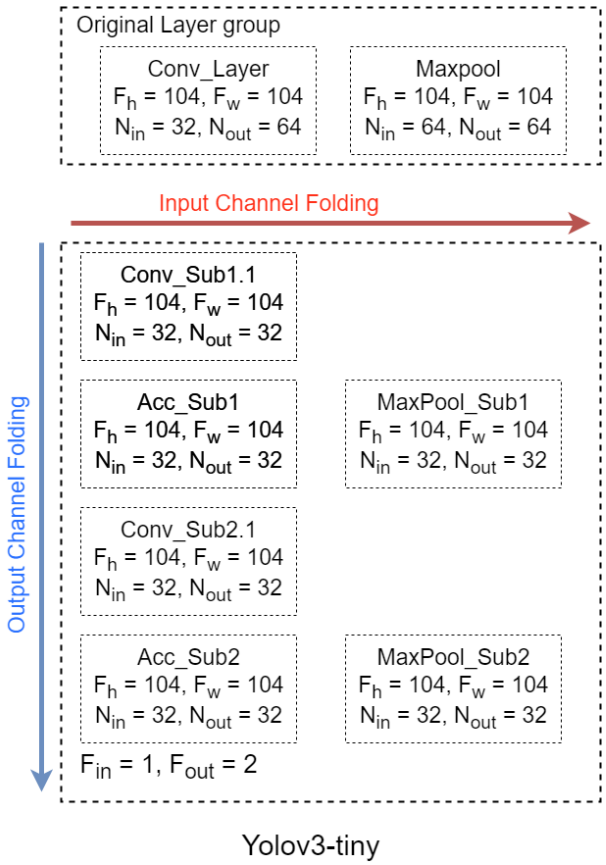


Figure 8.1: Yolov3-tiny Channel folding

resource utilization breakdown for Setup 1 is displayed in Table 8.13, indicating that approximately 55% of DSP resources are utilized for MACC operations in the YOLOv4-tiny model. The programmable logic portion of the Zynq-7000 SoC contains several DSP slices that can be utilized for arithmetic operations. Vivado HLS, Xilinx's high-level synthesis tool, supports the mapping of arithmetic operations onto DSP slices for the ARTIX family, which includes the Zynq-7000 series chips. Therefore, it is possible to utilize the DSP slices on the Zedboard through Vivado HLS. The 117 out of 121 DSPs come from the convolutional IP block where efficient pipelining and local memory partitioning caused all 9 MACC operations within multiply-accumulate batch instances to run in parallel. The local memory partition $P_{mem} = 8$ when set to $N_{max} = 32$ caused the optimal value of the initiation interval to be 10 clock cycles. Due to DMA constraint and 16-bit fixed point quantization, the number of such multiply-accumulate batches inferred in the design was equal to $\left\lceil \frac{4 \times N_{max}}{II_{conv}} \right\rceil = 13$ as N_{max} is set to 32. Therefore the total number of DSP inferred by convolutional IP alone is 117. Rest 4 DSPs comes from the Accumulation IP block which applies activation leaky activation across 4 channels in parallel when bia memory is partitioned cyclically as $p_{acc} = 2$. The *post_process* function inside the code applies the leaky activation across 4 channels thereby causing the design to instantiate the 4 different *post_process* functional units.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	4	-	-	-
Expression	-	0	0	6556	-
FIFO	0	-	160	896	-
Instance	0	117	10581	4037	-
Memory	121	-	0	0	0
Multiplexer	-	-	-	18463	-
Register	0	-	28709	384	-
Total	121	121	39,389	30336	0
Available	280	220	106400	53200	0
Utilization (%)	43	55	37	57	0

Table 8.13: Resource utilization block report: **setup** 1

Furthermore, most of the BRAM utilization (120 BRAM18k blocks) comes from the local memory for storing weights and from the line buffers which are used for storing inputs feature map values in convolutional IP blocks. As already discussed, $P_{mem} = 8$ is a more optimal value in terms of latency and resource utilization when $N_{max} = 32$. Rest 2 BRAM comes from the partitioning of the bias memory by a factor of 2 inside accumulation and activation IP block. Low utilization of BRAM_18K indicates that the accelerator is not heavily using the on-chip memory. This is the case as weights and inputs are stored in off-chip memory. The design has been optimized to use off-chip memory instead. It could also suggest that there is room for improvement in the design, by potentially reducing the amount of off-chip memory accesses and increasing the usage of on-chip memory by utilizing loop tiling techniques [40]. Loop tiling, also known as loop blocking, involves dividing a loop iteration space into smaller blocks or tiles. By processing a smaller block of data at a time, we can increase the locality of memory accesses, which can improve the cache and memory hierarchy performance. This can reduce the number of off-chip memory accesses and increase the usage of on-chip memory. Table 8.14 and 8.15 give the breakdown of the resource utilization at the functional level. Both IP, Convolutional and Accumulation IP, have top-level functions such as $Yolo\_Conv\_Top$ and $Yolo\_Acc\_Top$ and the rest of the components are the leaf cells which are called with the top-level module. The top-level module in Vivado can have a significant impact on resource utilization, as it is the module that integrates all other modules and components into the final design. Therefore their contribution is reflected in the resource utilization table of the design 8.13.

The conclusion that can be made from these two tables is firstly, BRAM utilization is very low. Optimizing the memory access time by loop tiling strategy can reduce the off-chip memory access. Secondly, the number of channels that can be processed in parallel is constrained to 32 in this design due to resource constraints. This can be increased in the future to 64 to process more channels at a time which can have a significant impact on DSP utilization.

COMPONENT	BRAM_18K	DSP48E	FF	LUT	URAM
Yolo Conv Top	120	117	37583	28404	0
Window Macc	0	9	737	112	0
Sliding Window	0	0	135	222	0
Post Process	0	0	0	169	0
Output Merge	0	0	198	781	0

Table 8.14: Resource utilization for convolution IP functions.

Component	BRAM_18K	DSP48E	FF	LUT	URAM
Yolo Acc	2	4	1237	1930	0
Post process	0	1	150	224	0

Table 8.15: Resource utilization for YOLOv4-tiny Accumulation and Activation block.

8.4.2 Resource Utilization Breakdown: setup 2

Table 8.16 shows overall resource utilization post-synthesis when all the layers except the route layer are hardware accelerated. The resource utilization shows better DSP utilization of about 75%. Moderate LUT RAM utilization is similar to setup 1 which is again due to a dynamically configurable architecture that has been employed, which takes the layer group as the fundamental unit. This layer group in the design has to reload the weights of each layer multiple times instead of storing them on the FPGAs before runtime. This is because the BRAMs on Zedboard are not large enough to hold all the weights.

Resource	Utilization	Available	Utilization %
LUT	29709	53200	55.84
LUTRAM	350	17400	2.01
FF	45915	106400	43.15
BRAM18k	149	280	53.21
DSP	166	220	75.45

Table 8.16: Resource Utilization: setup 2

8.5 Platform Comparison

The testing platform employed for the study is the Zedboard development kit that features the Xilinx XC7Z020-CLG484-1 SoC and has a 512 MB DDR3 [41]. The clock frequency for the programmable logic and processing system of the FPGA chip is 100 MHz and 667 MHz, respectively. There are 280 BRAM, 220 DSP, 106k FF, and 53k LUT resources accessible within the programmable logic. The clock frequency of an FPGA affects its performance. If the programmable logic (PL) has a higher clock frequency, then it can process data faster, leading to a higher throughput. However, this also means that the pipeline may need to be deeper to meet timing constraints. As for ARM processors, a higher frequency can result in a smaller instruction cycle, allowing software drivers to run faster. Table 8.17 shows the comparison of implemented architecture with the available literature on Zedboads and another

Xilinx-based platform.

	Ref [30]	**Ref [8]**	**Ref [1]**	**This Work**
Platform	Ultrascale+ZZCZU9EG	ZedBoard	ZedBoard	ZedBoard
PL frequency (MHz)	143	100	100	100
PS frequency (MHz)	-	667	667	667
BRAM_18K	384	65.5	185	149
DSP48E	839	19	160	166
LUT	139k	42k	25.9	29.7k
FF	124k	56.1	46.8	45.9k

Table 8.17: Platforms comparison of the proposed design with previous works

The ref [30] is similar to this work in terms of network architecture (YOLOv4-tiny) implemented. The IP core in Ref [30] has a matrix of vector functional units that share configurations among the same type, an integrated DMA for fast data transfers, heterogeneous stages, automatic ping-pong memories, and address generation units handling 6-level nested loops without software intervention. The custom MAC-based FUs are organized in a matrix structure to exploit Inter-FM, Intra-FM, and Inter-convolution parallelism, which enhances pixel and weight sharing. Here, the IP core speeds up the execution of YOLO, upsample, and max pool layers by running them together with the preceding convolutional layer. Additionally, the IP core accelerates the pre-CNN procedure and the drawing of the detections in the post-CNN procedure, unlike this design which does not accelerate pre-processing and post-processing but implements them inside PS only.

The difference in resource utilization is due to different design choices made. For example, in this particular design, the weights of each layer are not stored on the FPGAs before run-time but rather reloaded repeatedly. This is because the BRAMs on the Zedboard cannot store all the weights at once. To overcome this limitation, a dynamic and configurable architecture has been adopted, where layer groups are taken as the basic unit. In contrast, designs like Ref [30] deploy the entire network on hardware, which requires a lot of resources but reduces the transactions between hardware and off-chip memory. These designs use optimization techniques like tiling to split weights into batches and improve data locality, taking advantage of the availability of large-size BRAM.

8.6 Speed and Resource Efficiency

Latency is the primary focus in achieving real-time for embedded object detection. In this specific configuration, latency and throughput are interchangeable because each frame is processed in a strictly sequential manner. However, if frames are divided into a pipeline, high throughput does not necessarily translate to low latency.

The latency of the design is 3.3 sec for getting one image detected. The workload of YOLOv4-tiny is 6.845 GOPS. The throughput of this design is approximately equal to the:

$$\textbf{Throughput} = Workload \times FPS = 6.825 \times \frac{1}{3.3} = 2.05\,\text{GOPS/secs}$$

The latency difference between Ref [1] and this work has been explained earlier. In design [30], the entire network has been incorporated inside the hardware thereby decreasing the memory access times utilizing the on-chip memory. The issue with focusing solely on resource efficiency is that it overlooks the ability to scale those resources. The previous works apart from Ref [1] cited in this research were not adjustable in terms of balancing resources and performance. To achieve scalability, the design necessitates complex control logic and more multiplexing, which can be challenging. Moreover, scalability impacts design choices. For example, as discussed in chapter 6, interleaving channels allow for adjustable buffer size, but such decisions may not result in the most efficient performance.

In terms of area efficiency given by metric $\frac{\text{GOPS}}{\text{DSP}}$, $\frac{\text{GOPS}}{\text{KLUT}}$, and $\frac{\text{GOPS}}{\text{KFF}}$, this work shows better area efficiency as compared to Ref [8] It is crucial to take into account the range of scalability when evaluating different designs. For example, one design may exhibit excellent resource efficiency across the entire design space, but its tunability may be limited to a narrow range. These considerations explain some reasons that may not appear highly competitive compared to other research works discussed earlier in the text.

	Ref [30]	**Ref [8]**	**Ref [1]**	**This Work**
Latency (ms)	26	59385	532	3325
Throughput (GOPS)	212	0.113	10.45	2.05
Efficiency (GOPS/DSP)	0.16	0.005	0.065	0.012
Efficiency (GOPS/BRAM)	0.55	0.001	0.056	0.027
Efficiency (GOPS/kLUT)	1.52	0.002	0.40	0.06
Efficiency (GOPS/kFF)	1.69	0.002	0.22	0.044

Table 8.18: Speed and Efficiency w.r.t previous works

8.7 Power and Energy Efficiency

Power includes both on-chip and off-chip components, and for embedded applications, minimizing power consumption is generally desirable. To assess the effectiveness of a design, it is essential to consider both speed and power consumption, which can be evaluated using metrics such as energy per frame and power efficiency. Although a design may have high power consumption but run fast and use sleep mode when idle to save energy, it is still crucial to evaluate its energy per frame or power efficiency to determine its overall effectiveness.

The power consumption of the design is measured using a post-synthesis report from Vi-

vado which reported the power consumption as 2.46 W for the design as shown in table 8.19. Ref [30] is more efficient in terms of power efficiency. In terms of energy efficiency, Ref [8] is more efficient as it utilizes a more efficient architecture that works on the principle of Row stationary(RS) dataflow. RS dataflow is designed to minimize energy consumption associated with data movement on a spatial architecture. This is achieved by taking advantage of the local data reuse of filter weights and feature map pixels, which are activations, in high-dimensional convolutions, while also minimizing data movement of partial sum accumulations. Overall, this design is intended to be highly efficient in terms of energy consumption and can accommodate a wide range of CNN configurations.

	Ref [30]	**Ref [8]**	**Ref [1]**	**This Work**
Power (W)	4.5	2.32	3.36	2.46
Power efficiency (GOPS/W)	47.11	0.1092	3.11	0.833
Energy/Frame (J)	173	0.037	1.79	0.745

Table 8.19: Power comparison w.r.t previous works

8.8 Comparison with CPU and FPGA

Intel(R) Core(TM) i5-8250 CPU with a clock speed of 1.6 GHz and 8 GB DDR4 memory running at 2133MHz. During the test, only one core, or more specifically, one hyper-thread was utilized. The laptop runs on Windows OS. Throughout the rest of this chapter, this laptop configuration will be referred to as "Intel CPU".

The second test utilizes only the processing system of Zedboard, which is ARM-Cortex A9 operating on bare-metal at a speed of 667MHz, and the on-chip memory has a capacity of 512MB DDR3. This test platform is referred to as the "ARM CPU".

The testing program (YOLOv4-tiny) is tested as a hardware-software codesign with PS frequency of 667 MHz and PL with 100 MHz. According to Figure 8.2, the FPGA implementa-

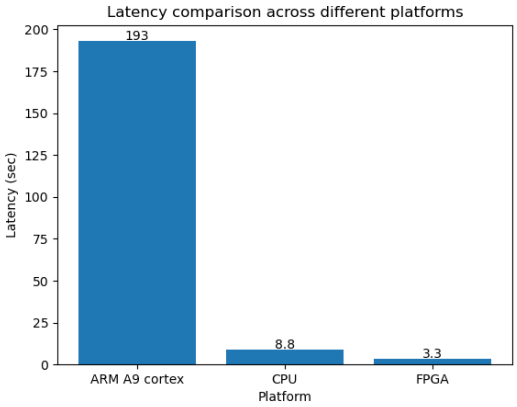


Figure 8.2: Latency comparison of 3 platforms

tion offers 55 and 2.6 times faster hardware acceleration compared to ARM and Intel CPUs, respectively. This significant difference in performance highlights the poor performance of Zedboard's processing system (PS) for this particular task.

Chapter 9

Conclusion and Recommendations

This chapter presents the conclusions by answering the research subquestions. The main research questions are discussed in section 9.1. The section 9.3 suggests further improvements that can be implemented in the design for further acceleration of the YOLOv4-tiny model.

9.1 Research subquestions

How can a YOLOv4-tiny be optimized at the software level before hardware accelerating the model?

Initially, the yolov4-tiny bare-metal model was developed in the Darknet framework, which used 32-bit floating-point values to store weights. This led to complex floating-point operations and an mAP of 44%. To reduce complexity, and memory requirements, and minimize the impact on accuracy, the weights were converted from 32-bit to 16-bit fixed-point values using a weight loss function. The weight quantization was performed keeping 8 bits for the integer and 8 bits for the fraction to handle overflow and quantization errors, as concluded in section 5.4. Using the `ap_fixed.h` library, the model was then converted from a floating-point model to a fixed-point model by changing the floating-point operations to fixed-point operations. Another optimization that was carried out was merging the batch-normalization layer into the convolution by transforming weights before run-time which is discussed in section 6.2.6. Both these optimization resulted in a 2% decrease in the mAP of the model from 0.440 mAP to 0.432 mAP. The reduction in weight precision also reduced the area and bandwidth requirements, as the weight size was decreased by a factor of 2.

What are the design decisions that are taken to accelerate the YOLOv4-tiny model?

The unified hardware accelerator has been designed which not only speeds up convolutional layers but other layers in the yolov4-tiny. The accelerator is based on the existing architecture [1] [12] that implements dynamically configurable architecture. The accelerator has been modified to include strided 2 convolutions along with the selection of line buffer sizes which has a direct impact on the BRAM inferred by the design. As the goal is to make a tunable

design that is efficient in terms of resource utilization and latency with minimal impact on accuracy, various design decisions were taken. Firstly, the design has been modified to integrate a 16-bit fixed point quantization scheme based on the weights of the yolov4-tiny model using `ap_fixed.h` library discussed in section 5.4. This leads to a reduction in the complexity of the operations, allowing for faster inference times. Secondly, the channel Interleaving strategy has been adopted which limits the size of the line buffers which has a direct impact on the number of BRAM blocks inferred by the design as discussed in section 6.2.7. Due to the channel interleaving, accumulation happens on pixels of different channels. This reduces the amount of data that needs to be read from or written to memory, as each memory access retrieves or stores pixels from multiple channels at once thereby reducing external memory bandwidth requirements. Additionally, the channel interleaving strategy helps in making the design more tunable as total buffer sizes are controlled by a maximum number of channels that an IP can process (N_{max}). As the yolov4-tiny network includes both 3×3 convolutions as well as $1x1$ convolutions, $1x1$ were converted to 3×3 by padding zeros to the kernel. This was done so that 1×1 can be made to share the same resource as 3×3 as intra-FM convolutions are already unrolled in the design.

How can an FPGA design for yolov4-tiny be made parameterisable? Given the parameters what values are optimal?

The design has been made more parameterisable by making the individual IP block designs more tunable. The optimal tunable value was selected on the basis of minimal resource utilization and latency. In a convolutional IP block, the tunable parameter P_{mem} controls the way the memory has been block partitioned. This in turn makes the design more parallel by enabling more convolution kernels to be convolved with input channels. $P_{mem} = 8$ was the more optimal choice as further increasing the tunable parameter range did not improve the initiation interval of the design from 10 clock cycles. Furthermore, the latency of the IP did not improve as the design inferred more multiplexers as compared to the simple expression when the tunable parameter range was increased beyond 8. The convolution IP block inferred 119 DSP cores with 54% utilization of the DSP148E block available on Zedboard. FF and LUT utilization for $P_{mem} = 8$ was 35% and 53% respectively. The accumulation and activation block in the design uses P_{acc} as the tunable parameter to cyclically partition the bias stored in bias memory. $P_{acc} = 2$ was found to be a more optimal design choice in terms of latency and resource utilization as it reduced the initiation interval of the design to 1. Furthermore, $P_{acc} > 2$ caused the array partition to be non-uniform which caused HLS to infer distributed RAM (FF and LUTs) in comparison to on-chip BRAM. For the max-pool IP block, P_{pool} parameter controlled the resource sharing of the window max function. It was found that the optimal value of the tunable parameter was $P_{pool} = 2$. Finally, in the Yolo Layer. P_{yolo} has been used as a tunable parameter to control the hardware associated with doing fixed point exponential sigmoid functions across the input channels. The optimal point for this tunable parameter is $P_{yolo} = 1$ as the resource utilization was minimal i.e. 3% and 5% of Flip-flop and LUT.

How can a YOLOv4-tiny accelerator be created using the Vitis unified software platform?

The Vitis unified software platform was utilized to develop a hardware-software codesign for the YOLOv4-tiny model, where all layers except for the route layer were accelerated using a pre-existing architecture from [1]. Hardware IP blocks for the convolutional, max-pool, upsample, and YOLO layers were designed using Vivado HLS, and custom datatypes, templates, and pragmas were used to create an appropriate description on for the accelerator. Pipelining and array partitioning was used for convolutional IP block to parallelize the design. The number of parallel instances of multiply-accumulate batch units was determined based on the partitioning of the 3D local memory storing the filter weights, which improved the design's throughput. The accumulation and activation IP module employed a cyclic array partition of bias memory and controlled the HLS tool's inference of on-chip BRAM instead of distributed RAM. The HLS allocation directive was used to optimize the performance of the Maxpool and YOLO layers. The max pool layer required additional hardware to select the maximum element in a 2×2 window across four input channels, while the YOLO layer performed a fixed-point exponential sigmoid function across four input channels. However, the resource requirements for these operations were shared using the HLS allocation pragma, and an optimal choice of latency with respect to resource utilization was made as discussed in chapter 6. The Vivado design suite is utilized to integrate all IP blocks. Once the bitstream is generated, the Vitis Application Project (VAP) and Platform Project (VPP) are created. The VAP consists of software source code and build scripts that are necessary to build the software application. In this case, the software driver discussed in chapter 7.1 is responsible for communication with the hardware developed through AXI4 interfaces within the Vitis environment. Vitis provides the Vitis application project for this purpose. The typological configuration of each layer, along with the input and output channel folding factor, is specified within the PS. The topological feature directly affects the layer's latency. The VPP contains the hardware design files, such as the Vivado HLS-generated IP blocks, platform specifications, and other required hardware components to implement the accelerator. The yolov4-tiny accelerator in this thesis design inside the Vitis unified software platform is able to make inferences at the rate of 3.3 sec/image.

Can a design space exploration be carried out for the YOLOv4-tiny model to find optimal design points which reach low latency with as few resources as possible?

Yes, a design space exploration can be carried out to determine the Pareto optimal points. variation in latency was calculated with respect to both DSP and BRAM units inferred by the design using the analysis done for all the IP blocks. Firstly, a Python script-based simulation analysis was done taking into account the effect of variation in tunable parameters as discussed in research subquestion 3 on the estimated latency of each hardware IP block after channel folding is applied as discussed in chapter 6. Furthermore, the model also takes into account

the variation of the DSP and BRAM resources with respect to the variation in the tunable parameter for each IP block as discussed in chapter 6. The model also takes into account the estimated latency of the PS which is responsible for sending typological features to the IP through the DMA interfaces, weights, biases, and transfer of accumulated data. All PS estimations are discussed in chapter 7. All the tunable parameters range were taken into account. Variation of the maximum number of channels that an IP can process N_{max} was kept as a factor of 4 between 4 to 64 channels. Convolution IP memory block partition factor P_{mem} was kept between 1 to 32. Similarly, P_{pool}, P_{yolo}, and P_{pool} range was kept between 1 to 4 due to the DMA constraint of processing 4 channels in parallel. Pareto optimal simulation points were obtained in section 7.5 which showed that DSP utilization and latency have an inverse relationship but the slope of the latency curve flattens which shows that latency is limited by extra factors such as bandwidth constraint and latency due to software driver implemented in PS. A much sparser distribution curve for BRAM was obtained with respect to latency which suggests that the HLS tool inferred more distributed RAM as compared to on-chip BRAM when the memory partition isn't big enough for the tool to infer BRAM18 blocks. Finally, 2 design points were tested on hardware to check the correctness of the estimations and it was found the deviation in latency was within 7% and resource utilization was within 5%.

9.2 Main research question

Given the answers to the research subquestion, the main research questions can be answered as:

Can a tuneable FPGA design be created for the deep-learning object detector YOLOv4tiny with possible opportunity for carrying out design space exploration?

Yes, a tuneable FPGA design can be created for the deep-learning object detector YOLOv4-tiny. The performance of each IP block is controlled by the tunable parameter associated with that IP block. For the convolutional IP block, p_{mem} tunable parameter controlled the way local 3D weight memory gets partitioned. In the design, it was found that 13 instances of the multiply accumulate batch units were inferred and each unit was able to do 3×3 convolutions in parallel which took 2 clock cycles. The throughput of the entire design is 2.05 GOPS/sec. The entire design setup infers a total of 149 BRAM18K blocks with 166 DPS148E units. The FF and LUT utilizations are about 43.15% and 55.84% with an inference time of about 3.3 secs/image.

The design is able to accelerate not only the convolutional layer but also other non-convolutional layers. The hardware-software codesign is more efficient in terms of speed than the PS-only design by a factor of 58. The max-pool layer execution time decreases by a factor of 8, Up-sample layer by a factor of 4.5, and the Yolo layers by a factor between 8 to 11. The total execution time of all the non-convolutional layers decreases by a factor of 3. Although the accelerator does not comply with the real-time performance requirements of 30 FPS, the ar-

chitecture provides a scalable framework whose performance can be tuned depending on the resources available in the target platform as shown in design space exploration. Comparing the design with the designs found in the literature shows an important difference in DSP usage. These designs map MAC operations to DSPs in contrast to the design presented in this work [8], Vivado HLS supports the mapping of arithmetic operations to the DSP units as opposed to Catapult which does not support the mapping of arithmetic operations to DSPs directly for the FPGA integrated into the ZedBoard [9]. The DSP utilization in this design is 8-fold as compared to the design compared to design [8] and 5-fold as compared to design [9]. This design takes about 2 clock cycles to do MAC operations which differs from the design in comparison but the latency improvement in this design is about 18-fold. This is due to the highly parallel architecture which can do $4 \times N_{max}$ convolutions utilizing intra-FM, inter-FM, and inter-output parallelism.

9.3 Recommendations

The conclusion presents a promising architecture that provides gains in both latencies as well as optimal resource utilization in the case when DMA constraints are set to 64 bits and $N_{max} = 32$ channels and fixed-point quantization of 16 bits. Features and modifications that are not implemented because of a lack of time are discussed separately in this section.

9.3.1 Testing on different Xilinx-based platforms

The current work provides a strong foundation for testing the given architecture on more resource-rich Xilinx platforms. The design space exploration discussed in chapter 7 shows that best case estimated latency of 2.467 secs/image. This inference rate is achieved when $N_{max} = 64$, $p_{mem} = 32$, $p_{acc} = 2$, $p_{pool} = 2$, $p_{yolo} = 2$. The DSP and BRAM estimation at these design points are 410 DSP418E units and 432 BRAM18K blocks respectively which is far more than what is available on the Zedboard. The ZedBoard has about 220 DSP units and 280 BRAM blocks. To support the above configuration, the DMA bandwidth supported by AXI has to be increased to 1024 bits which is the maximum bit-width supported by AXI. Given the 16-bit quantization, this will make the maximum channels to be processed parallelly to 64. So there is scope to improve the latency of inference if the design is tested with above mentioned tunable design parameters on resource-rich platforms like Zynq Ultrascale+ MPSoCs [42].

9.3.2 Reduced bit-width implementation

Based on the findings in this work, more opportunities for further acceleration can be explored. A further investigation should be conducted to explore data quantization techniques in order to reduce data bit-width. Techniques such as logarithmic compression or binarized transformation can be attractive options, particularly for layers that are responsible for extracting features. These techniques can help reduce computational complexity and memory requirements while maintaining acceptable accuracy. However, it is important to note that

for layers involved in predicting bounding boxes, it is generally preferable to retain a higher precision to ensure accurate localization and precise object detection.

9.3.3 Non-uniform channel interleaving

Current architecture takes into account uniform channel interleaving based on the maximum number of channels that an IP can process. Due to channel folding in each layer type, both input and output channels are folded with a factor of 32 thereby making channel interleaving uniform across the layer inputs and outputs. In terms of future work, there is potential to explore various implementation versions for each layer type, utilizing different strategies while maintaining uniform interfaces. For instance, different versions could involve channels being interleaved in a layer type or in another layer type channel interleaving is not implemented. By considering a larger design space, there is an opportunity to optimize each design point without significantly compromising the overall performance of the unified architecture. This approach acknowledges that there is no one-size-fits-all solution and that better results can be achieved by considering different design choices for each layer type.

Bibliography

[1] Z. Yu and C.-S. Bouganis, "A parameterisable FPGA-tailored architecture for YOLOv3-tiny," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2020, pp. 330–344.

[2] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 247–257.

[3] Z. Zou, Z. Shi, Y. Guo, and J. Ye, "Object Detection in 20 Years: A Survey," *ArXiv*, vol. abs/1905.05055, 2019.

[4] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, "Object Detection With Deep Learning: A Review," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, pp. 1–21, 01 2019.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[6] A. Bochkovskiy, C.-Y. Wang, and H.-y. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," 04 2020.

[7] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016, pp. 326–331.

[8] L. Heinsius, "Real-Time YOLOv4 FPGA Design with Catapult High-Level Synthesis," Master's thesis, University of Twente, 2021.

[9] M. Minnen, "CNN Accelerator Throughput Improvement using High-Level Synthesis for FPGA," Master's thesis, University of Twente, 2022.

[10] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174 692–174 722, 2020.

[11] "Deep sparse rectifier neural networks, author=Glorot, Xavier and Bordes, Antoine and Bengio, Yoshua," in *Proceedings of the fourteenth international conference on artificial*

intelligence and statistics. JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[12] Y. Zhewen, "A Dynamic Configurable FPGA Implementation of YOLOv3-tiny. Msc thesis, Analogue and Digital Integrated Circuit Design of Imperial College London," 2019.

[13] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. Proceeding of Machine Learning Research, 2015, pp. 448–456.

[14] V. Wadawadagi, "Metrics to use to evaluate deep learning object detectors." [Online]. Available: https://www.kdnuggets.com/2020/08/metrics-evaluate-deep-learning-object-detectors.html

[15] J. Redmon, "Darknet: Open source neural network in C," 2013. [Online]. Available: https://pjreddie.com/darknet/

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[18] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[19] ——, "YOLOv3: An Incremental Improvement," *arXiv preprint arXiv:1804.02767*, 04 2018.

[20] C. Guo, X.-l. Lv, Y. Zhang, and M.-l. Zhang, "Improved YOLOv4-tiny network for real-time electronic component detection," *Scientific Reports*, vol. 11, no. 1, pp. 1–13, 2021.

[21] 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, 2007.

[22] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[23] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

[24] C. Zhu, S. Han, H. Mao, and W. Dally, "Trained ternary quantization 5th Int," in *Conf. on Learning Representations, ICLR 2017 (Toulon, France, 24–26 April 2017)*, 2017.

[25] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International conference on machine learning*. Proceedings of Machine Learning Research, 2016, pp. 2849–2858.

[26] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, 2016, pp. 26–35.

[27] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[28] 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, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*. Springer, 2016, pp. 525–542.

[29] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[30] D. Pestana, P. R. Miranda, J. D. Lopes, R. P. Duarte, M. P. Véstias, H. C. Neto, and J. T. De Sousa, "A full featured configurable accelerator for object detection with YOLO," *IEEE Access*, vol. 9, pp. 75 864–75 877, 2021.

[31] P. Babu and E. Parthasarathy, "Hardware acceleration for object detection using YOLOv4 algorithm on Xilinx Zynq platform," *Journal of Real-Time Image Processing*, vol. 19, no. 5, pp. 931–940, 2022.

[32] 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 *Proceedings of the 2018 ACM/SIGDA International Symposium on field-programmable gate arrays*, 2018, pp. 31–40.

[33] Z. Li and J. Wang, "An improved algorithm for deep learning YOLO network based on Xilinx ZYNQ FPGA," in *2020 International Conference on Culture-oriented Science & Technology (ICCST)*. IEEE, 2020, pp. 447–451.

[34] V. Jain, N. Jadhav, and M. Verhelst, "Enabling real-time object detection on low cost FPGAs," *Journal of Real-Time Image Processing*, vol. 19, no. 1, pp. 217–229, 2022.

[35] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 575–580.

[36] Xilinx, "Vitis High-Level Synthesis User Guide," 2021. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Fixed-Point-Identifier-Summary?tocId=jgeN4rPFF_M6nxbDn1H5HA

[37] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 19–24.

[38] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," in *International conference on machine learning*. PMLR, 2019, pp. 7543–7552.

[39] X. Inc., "Vivado Design Suite—AXI Reference Guide, UG1037 (v4. 0)," 2017.

[40] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead," *IEEE Access*, vol. PP, pp. 1–1, 11 2020.

[41] "Avnet ZedBoard." [Online]. Available: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/

[42] "SDSOC development environment, https://www.xilinx.com/products/design-tools/legacy-tools/sdsoc.html." [Online]. Available: https://www.xilinx.com/products/design-tools/legacy-tools/sdsoc.html

Appendix A

Yolov4-tiny Details

A.1 Weight Distribution of yolov4-tiny

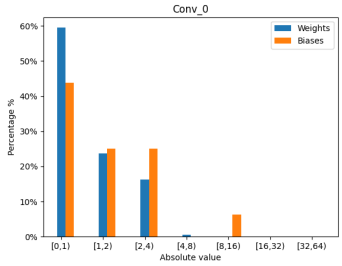


Figure A.1: conv0

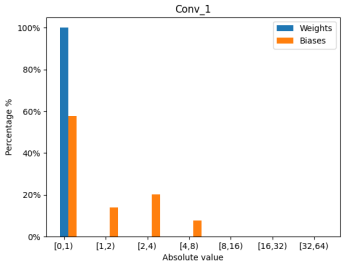


Figure A.2: conv1

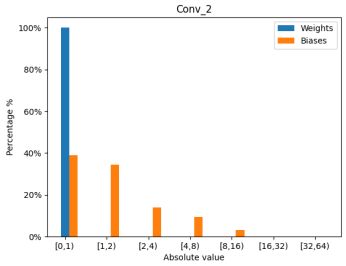


Figure A.3: conv2

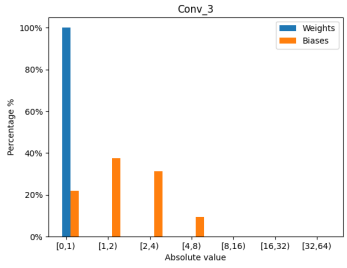


Figure A.4: conv3

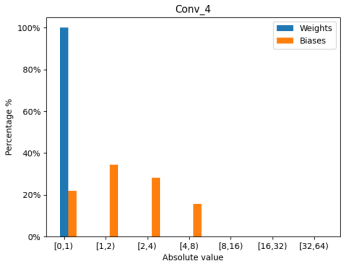


Figure A.5: conv4

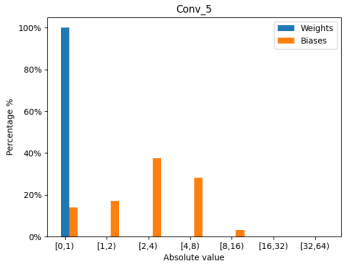


Figure A.6: conv5

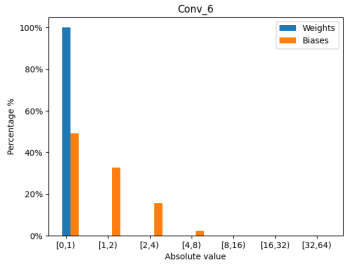


Figure A.7: conv6

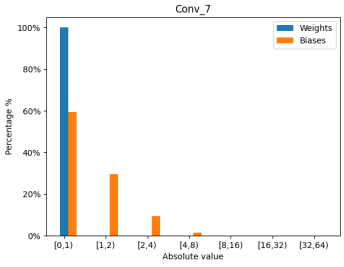


Figure A.8: conv7

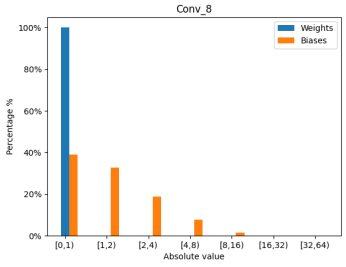
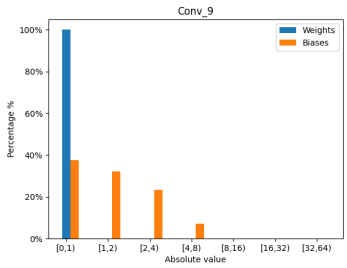
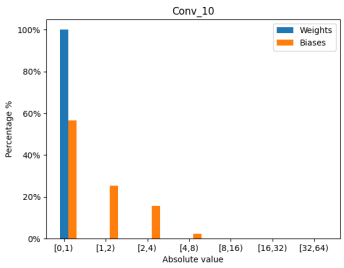
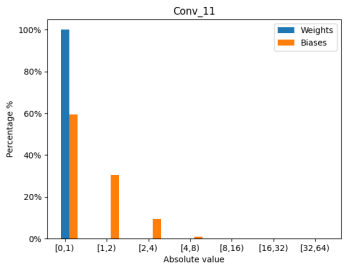


Figure A.9: conv8







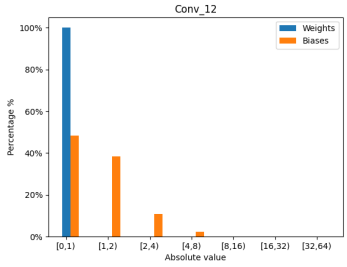


Figure A.13: conv12

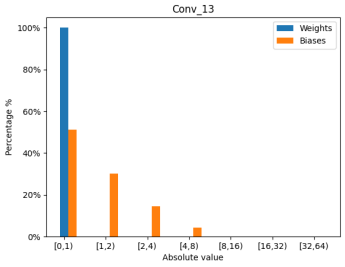


Figure A.14: conv13

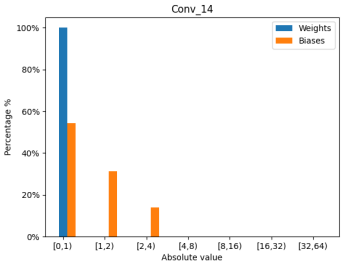


Figure A.15: conv14

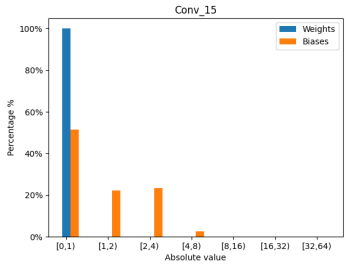


Figure A.16: conv15

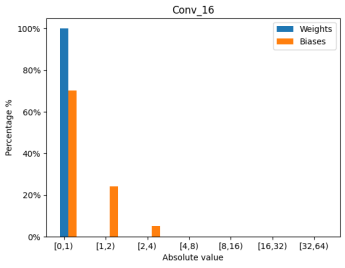


Figure A.17: conv16

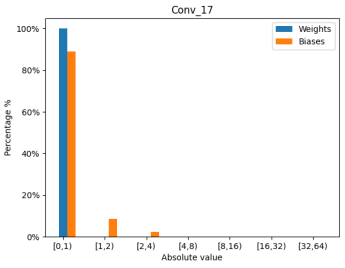


Figure A.18: conv17

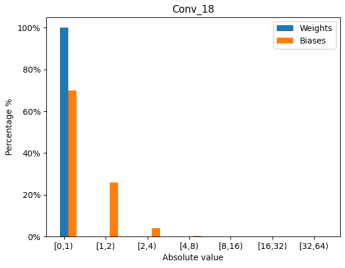


Figure A.19: conv18

Figure A.20: Weights Distribution for yolov4-tiny

A.2 Data Distribution of yolov4-tiny

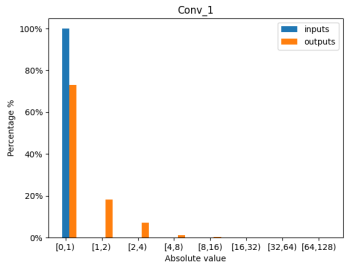


Figure A.21: conv1

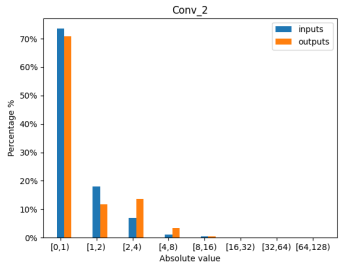


Figure A.22: conv2

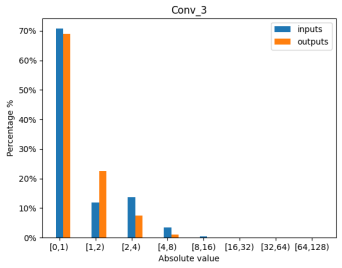


Figure A.23: conv3

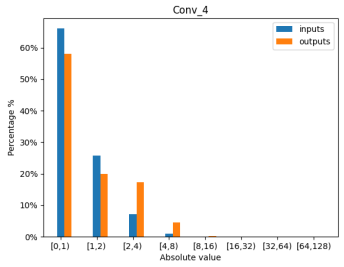


Figure A.24: conv4

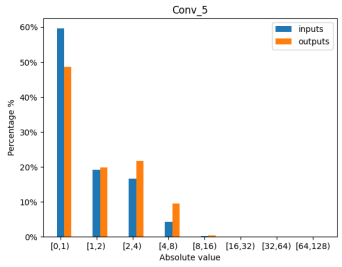


Figure A.25: conv5

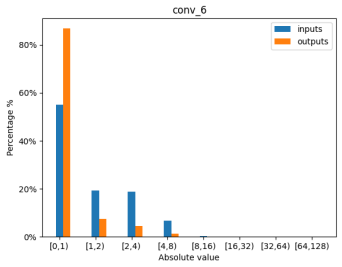


Figure A.26: conv6

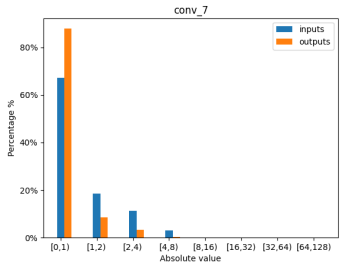


Figure A.27: conv7

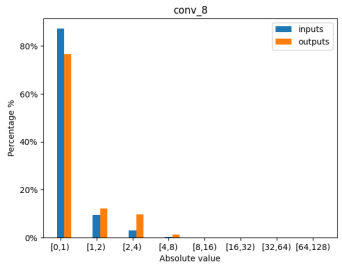


Figure A.28: conv8

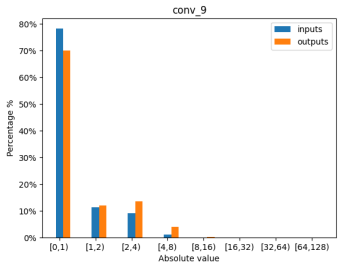


Figure A.29: conv9

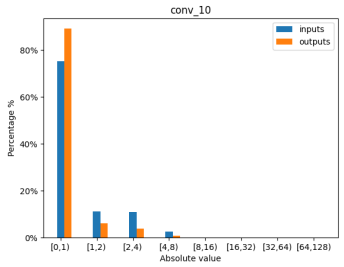


Figure A.30: conv10

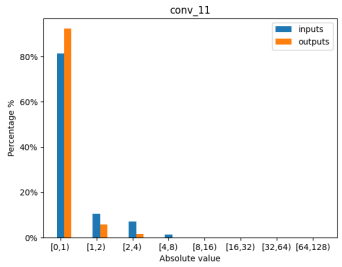


Figure A.31: conv11

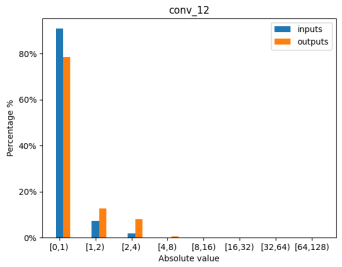


Figure A.32: conv12

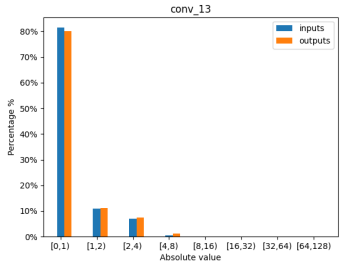


Figure A.33: conv13

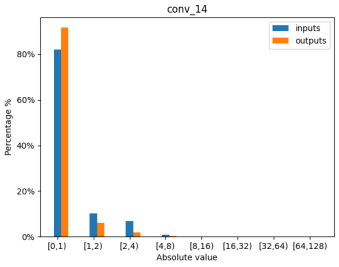


Figure A.34: conv14

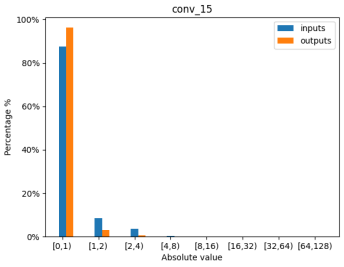


Figure A.35: conv15

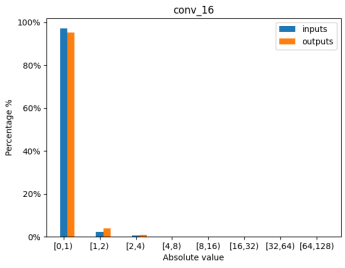


Figure A.36: conv16

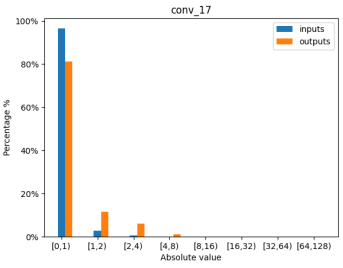


Figure A.37: conv17

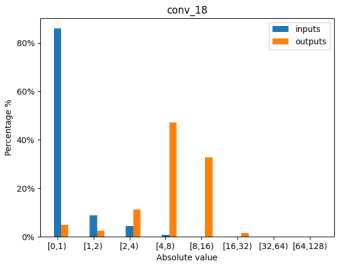


Figure A.38: conv18

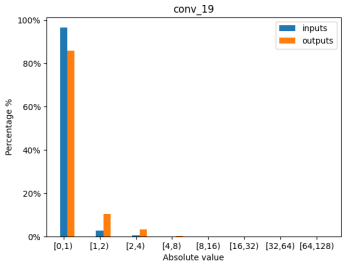


Figure A.39: conv19

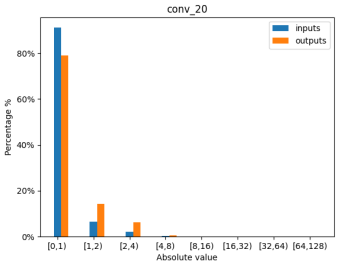


Figure A.40: conv20

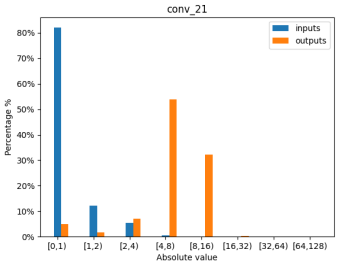
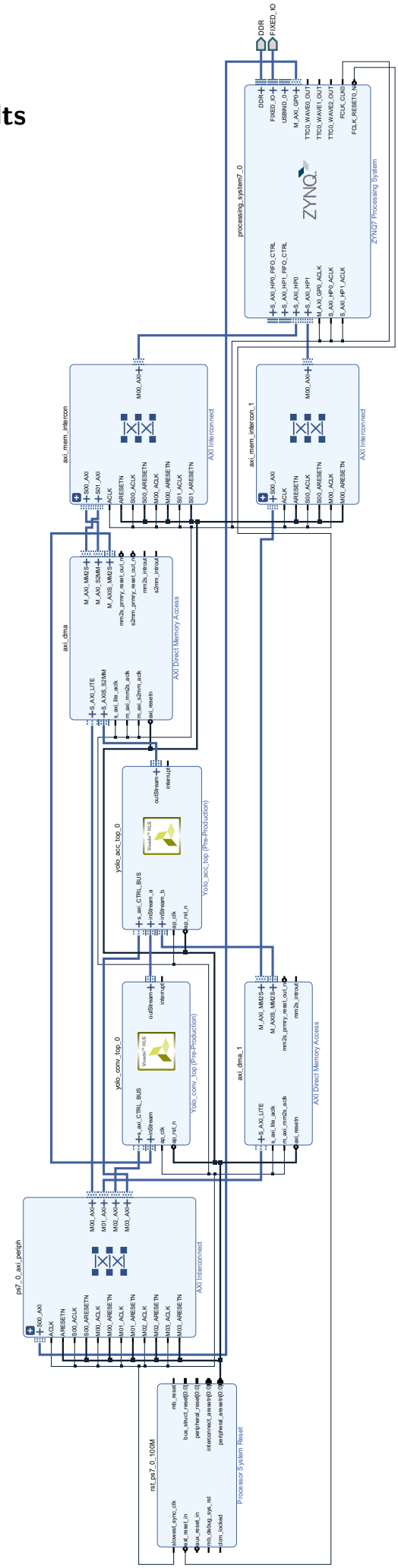


Figure A.41: conv21

Figure A.42: Data Distribution for yolov4-tiny

A.3 System Results

A.3.1 Setup 1



A.3.2 Setup 2