

國立臺北大學 通訊工程學系

碩士論文

指導教授:白宏達 教授

基於 PYNQ-ZU 之 YOLOv9 卷積層加速設計

**Design and Implementation of YOLOv9 Convolutional  
Layer Acceleration on PYNQ-ZU**

研究生:劉哲安

中華民國一百一十四年六月十四日

# ABSTRACT

With the rapid advancement of Artificial Intelligence (AI) and Deep Learning technologies, an increasing number of technology companies are allocating significant Resources to related applications. Among these, image recognition has become a major area of focus in both research and industry due to its high practicality. It is widely used in autonomous driving, security surveillance, and smart healthcare. Among various object detection algorithms, YOLO (You Only Look Once) stands out as one of the most widely adopted solutions, offering a combination of high accuracy and real-time processing capability.

This study utilizes the PYNQ-ZU board as the hardware platform to accelerate Convolution operations in the YOLOv9-tiny model by leveraging the parallel computing capabilities of FPGA. During development, Xilinx's Vitis HLS and Vivado tools are employed. HLS is responsible for implementing core functionalities, managing data precision, and allocating hardware Resources, while Vivado handles backend routing and Resource analysis. To reduce unnecessary consumption of on-Chip memory Resources, the Design optimizes data Precision and memory usage. Additionally, a Resource-sharing strategy is proposed to minimize redundant data storage. These optimizations enable a noticeable performance improvement while maintaining comparable recognition accuracy, even with a slight reduction in numerical precision.

# 致謝

在這段研究所的旅程中，我最想感謝的人，就是我的指導教授白宏達老師。從最一開始對研究感到茫然，到後來逐漸找到方向，老師一直都耐心地在旁引導我、鼓勵我。在這篇論文的研究過程中，無論是架構的規劃、實驗的方法，還是遇到困難時的調整方向，老師總是會用清楚的邏輯幫我釐清思路，給予我具體而中肯的建議。特別是在遇到挫折、做實驗失敗的時候，老師也會提醒我放慢腳步，重新檢視每個細節，這樣的鼓勵讓我在壓力中依然能堅持下去。

除了在學術上的幫助，老師平時也總是願意花時間與學生討論，分享自己在研究與人生中的經驗。這樣的互動讓我不只是學會了怎麼做研究，更學會了怎麼思考問題、怎麼面對挑戰。我很幸運能遇到這麼願意傾聽、願意陪著學生一起進步的老師。這段時間的成長，不只是因為完成了一個研究，而是因為有老師一路的支持與信任，讓我能走到今天。再次由衷感謝白宏達老師，這份指導與陪伴，對我來說是一輩子都會記得的事。

# Contents

ABSTRACT .....	i
致謝 .....	ii
Contents .....	iii
List of Figure .....	v
Chapter 1: Introduction .....	1
1.1 Motivation .....	1
1.2 Direction and Objectives .....	2
1.3 Organization .....	3
Chapter 2: Related Work .....	5
2.1 CNN .....	5
2.1.1 Introduction to CNN .....	5
2.1.2 Convolutional layer .....	7
2.1.3 Max Pooling and Average Pooling .....	8
2.2 YOLO .....	10
2.2.1 Introduction to YOLO .....	10
2.2.2 Convolutions in YOLO .....	11
2.2.3 YOLOv9-tiny Architecture .....	15
2.3 Overview of Vitis HLS, Vivado, and PYNQ-ZU .....	17
2.3.1 Vitis HLS .....	17
2.3.2 Vivado Design Suite .....	18
2.3.3 PYNQ-ZU .....	18
Chapter 3: Hardware/Software Co-Design .....	20
3.1 Environment Configuration .....	20
3.2 Analysis and Design of Parallel Processing .....	24
3.2.1 Convolution Mathematical Expression .....	24

3.2.2	Intuitive Parallel Processing Architecture .....	31
3.2.3	EPIC: Enhanced Partition for Independent Convolution.....	34
3.3	Shared Convolutional IP Design .....	38
3.3.1	Reasons for Sharing Convolution IP .....	38
3.3.2	Shared Convolution IP .....	38
3.3.3	Analysis of Resource Bottlenecks .....	41
3.3.4	Optimization Strategies for Resource Bottlenecks .....	43
Chapter 4: Research Results .....		47
4.1	Single-Layer Convolution Acceleration Performance .....	47
4.1.1	DATAFLOW Architecture Performance .....	47
4.1.2	EPIC Architecture Performance .....	48
4.2	Impact of Convolution Layer Sharing on Resource Utilization .....	50
4.2.1	Resource Savings Achieved Using ap_fixed<a, b> .....	50
4.2.2	Resource Optimization and TIP Sharing Analysis .....	51
4.3	Summary of Execution Results.....	56
Chapter 5: Conclusion and Future Work.....		60
5.1	Conclusion.....	60
5.2	Future Work.....	61

# List of Figure

Figure2- 1 Traditional Neural Network Processing Method .....	6
Figure2- 2 Convolutional Neural Network Processing Method.....	6
Figure2- 3 LeNet-5 Architecture [5].....	6
Figure2- 4 Illustration of Input Matrix and Kernel.....	7
Figure2- 5 Convolution Process with Stride = 1 .....	8
Figure2- 6 Max Pooling Process.....	9
Figure2- 7 Average Pooling Process.....	9
Figure2- 8 Convolution Process with Padding = 1 .....	13
Figure2- 9 Architecture of YOLOv9-tiny .....	15
Figure2- 10 Internal Module of YOLOv9-tiny (1) .....	16
Figure2- 11 Internal Module of YOLOv9-tiny (2) .....	16
Figure2- 12 Workflow of Vitis HLS Conversion .....	17
Figure2- 13 Examples of Hardware Control Directives in Vitis HLS.....	18
Figure3- 1 Vitis HLS Interface Layout e .....	20
Figure3- 2 Steps for Importing IP into Vivado.....	21
Figure3- 3 Usage of Custom IP in Vivado .....	21
Figure3- 4 Finalized IP Connection Diagram.....	22
Figure3- 5 Workflow for Generating .bit and .hwh Files.....	22
Figure3- 6 Viewing FPGA Hardware Resource Utilization .....	23

Figure3- 7 Convolution Computation Structure .....	25
Figure3- 8 Intuitive Parallel Processing Architecture .....	33
Figure3- 9 EPIC Architecture.....	35
Figure3- 10 Parallel Processing Module .....	36
Figure3- 11 Shared Convolution Architecture.....	40
Figure3- 12 Input and Weight Division Module .....	41
Figure3- 13 Resource Utilization of Multiple Convolution IP (Float32).....	42
Figure3- 14 Example of ap_fixed<3,2> .....	45
Figure4- 1 Pre-Optimization Architectural Issues .....	47
Figure4- 2 Results of DATAFLOW Optimization .....	48
Figure4- 3 Comparison of EPIC and Conv2d Performance .....	49
Figure4- 4 Hardware Resource Utilization: EPIC and Intuitive Designs .....	49
Figure4- 5 Hardware Resource Comparison: float32 vs. ap_fixed<18,6> ....	50
Figure4- 6 Distribution of Convolution Sizes in YOLOv9-tiny .....	51
Figure4- 7 Maximum Hardware Resource Usage for float32 .....	52
Figure4- 8 Maximum Hardware Resource Usage for ap_fixed<18,6> .....	53
Figure4- 9 Maximum Resource Usage Using LUTRAM.....	53
Figure4- 10 Comparison of BRAM Usage Among Different TIP Designs ...	54
Figure4- 11 Comparison of LUTRAM Usage Among Different TIP Designs	55
Figure4- 12 Final FPGA Hardware Resource Utilization .....	55
Figure4- 13 TIP Integration Results in Vivado .....	55
Figure4- 14 Performance and standard Deviation in 84 Conv Layers.....	57

Figure4- 15 mAP Under Different Precision Levels (coco128).....	57
Figure4- 16 Recognition Results with float32 Precision .....	58
Figure4- 17 Recognition Results with ap_fixed<18,6> Precision.....	58
Figure4- 18 Recognition Results with ap_fixed<17,6> Precision.....	59
Figure4- 19 Recognition Results with ap_fixed<16,6> Precision.....	59





# Chapter 1: Introduction

## 1.1 Motivation

In recent years, Deep Learning and Artificial Intelligence (AI) have emerged as focal points for investment among major technology companies. Among the various applications of these technologies, image recognition has garnered significant attention due to its wide-ranging value. For example, in the electric vehicle sector, intelligent driving technologies rely on image recognition for road condition analysis and object detection, laying the groundwork for autonomous driving. In addition, smart glasses can accurately identify objects in the user's line of sight through advanced image recognition, showcasing their immense potential in everyday applications.

Among the many image recognition algorithms, YOLO (You Only Look Once) has become one of the most widely adopted and recognized due to its high efficiency and accuracy. Since its initial release, YOLO has undergone numerous iterations. This study focuses on YOLOv9 [1], the latest version at the time of research. When performing edge computing on resource-constrained devices, YOLO is often executed using CPUs based on the ARM (Advanced RISC Machine) architecture. However, this results in slower processing speeds, which opens the door for FPGA (Field Programmable Gate Array) solutions.

FPGA offers customizable, parallel computation capabilities, making it highly suitable for offloading and accelerating complex, repetitive operations from the CPU. This aligns well with the needs of edge devices [2]. Furthermore, compared to ASIC (Application-Specific Integrated Circuit), FPGA provides rapid reconfigurability, allowing users to adapt to different hardware requirements and evolving algorithmic

structures. If future updates to YOLO involve changes to its computational flow, FPGA can be quickly reprogrammed to adapt.

The convolution operation, a core computational unit in YOLO, is particularly well-suited for FPGA acceleration [3] and is extensively used across many domains. For instance, Convolutional Neural Network (CNN) models are fundamental to modern image recognition tasks, and YOLO is based on such architecture. In recent years, the AI company OpenAI has applied CNN technology in its CLIP model, which has further been incorporated into ChatGPT [4], demonstrating the continued relevance and impact of convolution-based methods. As such, this technology presents significant potential for future development.

## 1.2 Direction and Objectives

This research aims to accelerate the most time-consuming component of YOLO convolution operation through FPGA-based hardware computation. When executed on a CPU (Central Processing Unit), YOLO encounters a major bottleneck due to the large number of convolution operations required. Convolution is inherently parallelizable, making FPGA an ideal platform to improve computational efficiency.

In this study, the PYNQ-ZU development board is used as an experimental platform. The design flow involves utilizing Vitis HLS, a development tool by Xilinx, to write the system in C++ and convert it into RTL (Register Transfer Level) code using Verilog or VHDL. The resulting RTL files are then compiled using Vivado to generate .bit and .hwh files for deployment on the PYNQ-ZU board. The primary goal is to offload part of the convolution computation in YOLOv9 to the FPGA, leveraging its parallel processing capabilities for acceleration.

The objectives of this research are as follows:

- Utilize the parallelism of FPGA to accelerate the convolutional operations in YOLOv9.
- Achieve improved performance under the same hardware resources or reduce hardware resource usage while maintaining the same level of performance.
- Ensure that the accuracy of object detection remains comparable to the original software model.

## 1.3 Organization

This thesis is divided into five main chapters. Chapter 2 provides a basic introduction to CNN and a detailed analysis of the convolutional layer, which is the key computational component of CNN. It then introduces the YOLO model adopted in this research, with YOLOv9, the latest version at the beginning of the study, serving as the research target. The chapter concludes with an overview of the development tools used in this study, including Vitis HLS and Vivado Design Suite. Chapter 3 focuses on the core of this research: how to utilize the parallel processing characteristics of FPGA to accelerate convolution operations. It starts with a mathematical model and first builds an intuitive parallel computing architecture. After iterative refinement, the final hardware design architecture is determined. Once the architecture is finalized, further discussion is provided on how to enhance the utilization of FPGA hardware resources through resource-sharing strategies. This chapter also analyzes the primary performance bottlenecks of the design, explores the influencing factors, and proposes corresponding optimization strategies to improve computational performance and hardware efficiency. Chapter 4 begins by presenting the unoptimized convolution performance on FPGA as a baseline for comparison. It then analyzes resource utilization before and after

optimization, explains how resources are allocated, and demonstrates the effectiveness of the resource-sharing strategy, highlighting its impact on FPGA hardware resource efficiency. At the end of Chapter 4, the FPGA-accelerated YOLO model is compared with the pure software version in terms of accuracy and performance, and the differences in standard deviation between the two versions are evaluated to assess the practical acceleration effect and advantages of the proposed approach. Chapter 5 concludes the findings of this research and explores potential future optimization directions to further improve the performance and resource efficiency of YOLO acceleration on FPGA.



# Chapter 2: Related Work

## 2.1 CNN

### 2.1.1 Introduction to CNN

CNN is a deep learning model specifically designed to process two-dimensional structured data, such as images [5]. Traditional neural networks typically flatten images into one-dimensional vectors before processing. However, this flattening process results in the loss of spatial structural information within the image, which can degrade the model's performance, as illustrated in Figure 2-1. In contrast, CNN use convolutional operations to extract local features from the image, preserving the spatial structure while significantly reducing the number of parameters and computational complexity during training, as shown in Figure 2-2.

The fundamental architecture of CNN consists of an input layer, convolutional layers, pooling layers, and fully connected layers. The convolutional layer is the core component of CNN, employing multiple filters to extract features from the input image and generate corresponding feature maps. The main purpose of the pooling layer is to reduce the dimensionality of the feature maps, thereby enhancing computational efficiency while retaining key information. Additionally, pooling layers improve the model's robustness to minor spatial shifts in the input image ensuring that slightly displaced features can still be recognized. Common pooling methods include max pooling and average pooling. The fully connected layer maps the high-level extracted features to the final output, enabling classification or regression tasks. For instance, the classic LeNet-5 architecture [6]

(Figure 2-3) consists of a feature extraction part and a classification part (fully connected layers), designed specifically for handwritten digit recognition.

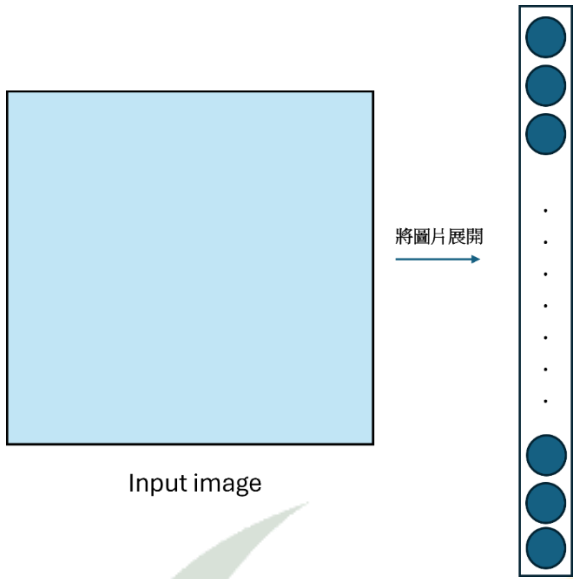


Figure2- 1 Traditional Neural Network Processing Method

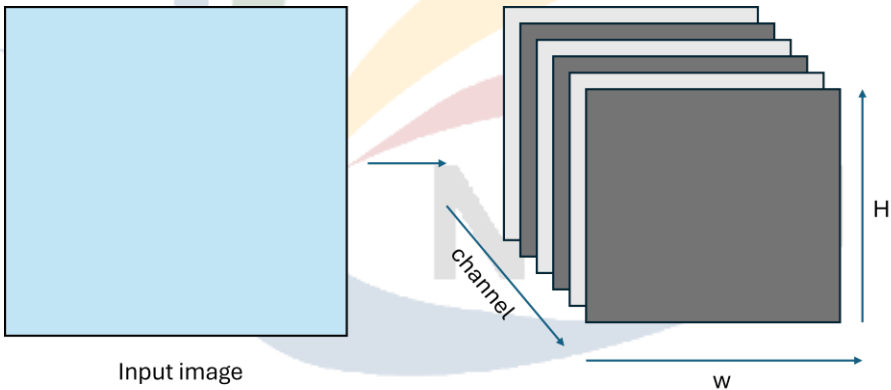


Figure2- 2 Convolutional Neural Network Processing Method

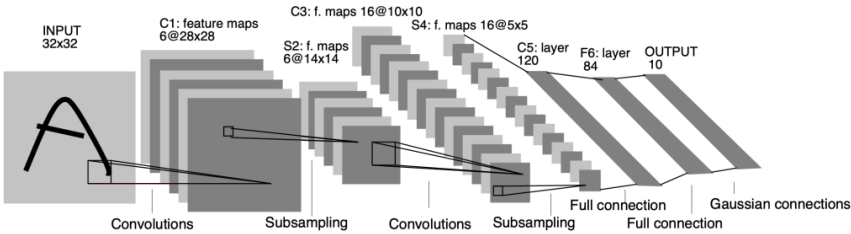


Figure2- 3 LeNet-5 Architecture [5]

## 2.1.2 Convolutional layer

The primary function of a convolutional layer is to perform element-wise multiplication and summation between the input matrix and a trained kernel, as illustrated in Figure 2-4. Through a sliding operation, the kernel moves across the input matrix to extract local features by computing the sum of element-wise products at each position a process demonstrated in Figure 2-5. This method enables the model to efficiently and comprehensively capture important local patterns from the input image, resulting in informative feature maps that enhance the neural network's recognition capability and computational efficiency. This process is known as two-dimensional convolution (2D convolution), and its mathematical expression is given in Equation (2.1), where  $X$  denotes the input feature map,  $\Omega$  represents the kernel, and  $Y$  indicates the output feature map. The indices  $i$  and  $j$  correspond to the width and height dimensions of the input, respectively.

$$Y[i,j] = \sum_{K_1=0}^{K_1-1} \sum_{K_2=0}^{K_2-1} \Omega[K_1, K_2] X[i + K_1, j + K_2] \quad (2.1)$$

1	1	1	0
0	1	1	1
0	0	1	1
0	0	1	1
0	1	1	0

5X4 input

1	0	1
0	1	0
1	0	1

3X3 kernel

Figure2- 4 Illustration of Input Matrix and Kernel

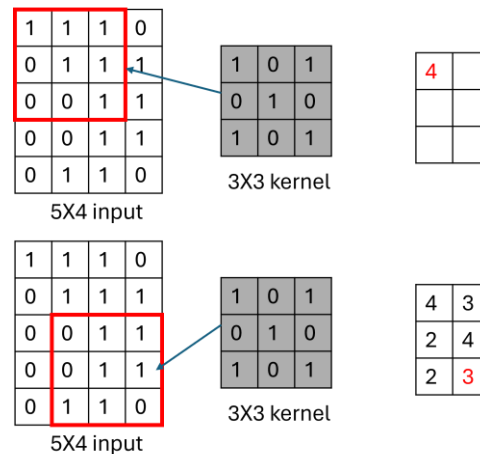


Figure2- 5 Convolution Process with Stride = 1

### 2.1.3 Max Pooling and Average Pooling

Max pooling and Average pooling are commonly used dimensionality reduction techniques in CNN. Their primary purpose is to reduce the size of feature maps, preserve essential features, and minimize computational load on hardware. In max pooling, the pooling window slides across the feature map at a predefined stride, and the maximum value within each window is selected as the output for the corresponding position. This approach emphasizes the most salient features in the image and offers a degree of robustness against noise, making it particularly suitable for edge detection or extraction of features in regions with abrupt changes. The operation is illustrated in Figure 2-6. In contrast, average pooling computes the average value of all elements within the pooling window and outputs that as the result. This method retains more of the overall background information from the region, helps smooth the feature map, and reduces the risk of overfitting. Its operation is shown in Figure 2-7.



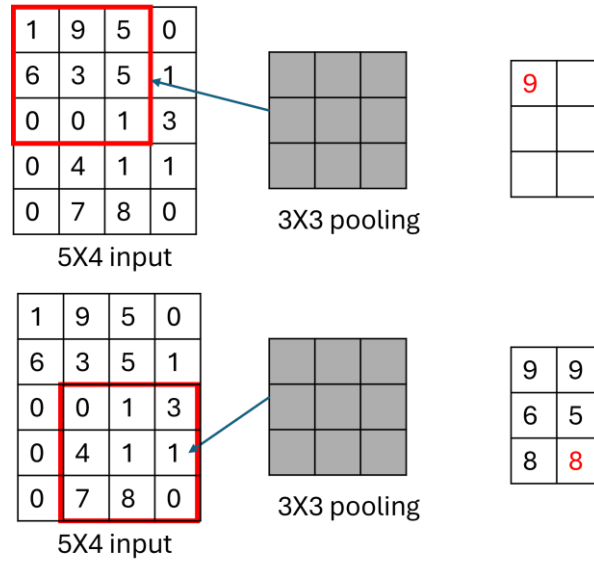


Figure2- 6 Max Pooling Process

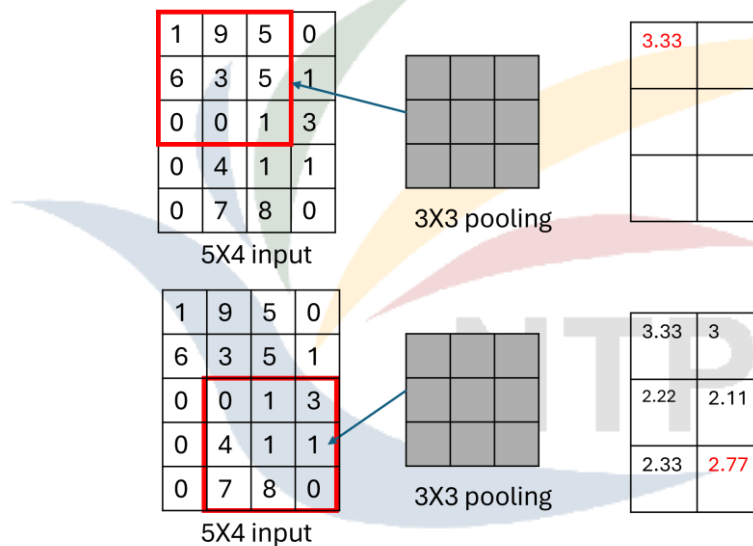


Figure2- 7 Average Pooling Process

Compared to average pooling, max pooling is more commonly used in deep learning models, particularly in object recognition tasks. This is mainly because max pooling preserves the most representative local information during feature extraction, rather than simply averaging all pixel values within a region. Since the core objective of object recognition is to identify key features of the target object, max pooling effectively highlights the most prominent feature regions in the image. This enhances the model's

ability to recognize primary targets while avoiding excessive smoothing or weakening of features that can occur with average pooling.

## 2.2 YOLO

### 2.2.1 Introduction to YOLO

YOLO is an advanced real-time object detection algorithm that has significantly reshaped the field of computer vision. Since its initial introduction by Joseph Redmon et al. in 2015 [7], YOLO has continuously evolved, setting new benchmarks for detection speed and efficiency. YOLOv3, released in 2018 [8], addressed the shortcomings of earlier versions, particularly in detecting small objects and managing crowded scenes. Subsequent iterations brought further advancements, notably YOLOv5 by Ultralytics [9], which improved overall performance, stabilized detection results, and enhanced ease of deployment. In 2020, YOLOv7 integrated technologies such as RepConv, achieving substantial accuracy gains without sacrificing inference speed, making it a representative version that balances real-time performance with high precision [10]. These continuous improvements have led to greater accuracy and practicality, cementing YOLO as the foundation for many real-time object detection applications.

Compared to traditional object detection algorithms, YOLO strikes an effective balance between accuracy and speed, while offering extensive object recognition capabilities across diverse application domains. For example, in UAVs and autonomous driving, YOLO can be used for real-time detection of pedestrians, vehicles, and traffic signs to ensure safe navigation [11]; in medical imaging, it assists physicians in identifying tumors or abnormal regions, thereby enhancing diagnostic efficiency and

accuracy [12]; in surveillance systems, YOLO enables automatic monitoring of abnormal behaviors or specific targets to improve security and anomaly detection [13]; and in smart glasses, it supports real-time object recognition and navigation for outdoor movement [14]. With its exceptional computational efficiency and wide applicability, YOLO has become a critical technology in object detection and continues to drive the advancement and practical deployment of deep learning across various industries.

## 2.2.2 Convolutions in YOLO

As the core operation within the YOLO architecture, convolution plays a crucial role in its performance. To implement 2D convolution in Python, YOLO utilizes the widely adopted Conv2d function from PyTorch [8], which is one of the most used methods for 2D convolution in Python. This function includes several important parameters, such as `in_channels`, `out_channels`, `kernel_size`, `stride`, and `padding`:

- i. `in_channels`: the channel count of the input feature map, typically 3 in YOLO to correspond to the RGB color channels.
- ii. `out_channels`: the channel count of the output feature map.
- iii. `Kernel_size`: The size of kernel is defined as a tuple  $(K_1, K_2)$ .
- iv. `stride`: The distance of the kernel moves across the feature map in a single step, with a default value of 1.
- v. `Padding` : The default padding value is 0, but when  $K_1=K_2>1$ , appropriate padding is applied to the edges of the input feature map to maintain the same output size.

The formats of the input and output feature maps are structured as follows. The output feature map's height  $H_\ell$  and width  $W_\ell$  are calculated using the equations shown in (2.1) [15]. In these equations, the value of *dilation* is set to 1 by default. It can be observed

that the output dimensions are affected by padding, kernel\_size, and stride. When the computed result is a non-integer, the value is rounded up to the nearest integer to ensure the integrity of the output feature map. Both the input and output feature maps consist of three primary dimensions:  $C$  (Channel),  $H$  (Height), and  $W$  (Width), where the subscript  $\ell$  at the lower right corner indicates the index of the current convolutional layer.

$$\text{Input: } (N, C_{\ell-1}, H_{\ell-1}, W_{\ell-1}) , N=1$$

$$\text{Output: } (N, C_{\ell}, H_{\ell}, W_{\ell}) , N=1$$

- $H$ : Represents the vertical direction of the feature map, that is, the j-axis.
- $W$ : Represents the horizontal direction of the feature map, that is, the i-axis.
- $C$ : Indicates how many  $H \times W$  matrices are present in this feature map.

$$\begin{aligned} W_{\ell} &= \left\lceil \frac{W_{\ell-1} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rceil \\ H_{\ell} &= \left\lceil \frac{H_{\ell-1} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rceil \end{aligned} \quad (2.2)$$

The relationship between  $H$ ,  $W$ , and  $C$  is illustrated in Figure 2-8. On the left side of the figure, the input feature map has a height  $H_{\ell-1}=5$ , a width  $W_{\ell-1}=4$ , and a channel count of  $C_{\ell-1}$ . This means the input feature map consists of  $C_{\ell-1}$  matrices of size  $5 \times 4$ . After performing a convolution operation using a  $3 \times 3$  kernel, the resulting output feature map has dimensions  $H_{\ell}=5$  and  $W_{\ell}=4$ , and the number of channels becomes  $C_{\ell}$ . Thus, the output feature map contains  $C_{\ell}$  matrices of size  $5 \times 4$ , each corresponding to a convolutional output. The three-dimensional relationship between  $H$ ,  $W$ , and  $C$  can also be seen in the right-hand side of Figure 2-2, where Channel corresponds to  $C$ , Height to  $H$ , and Width to  $W$ .

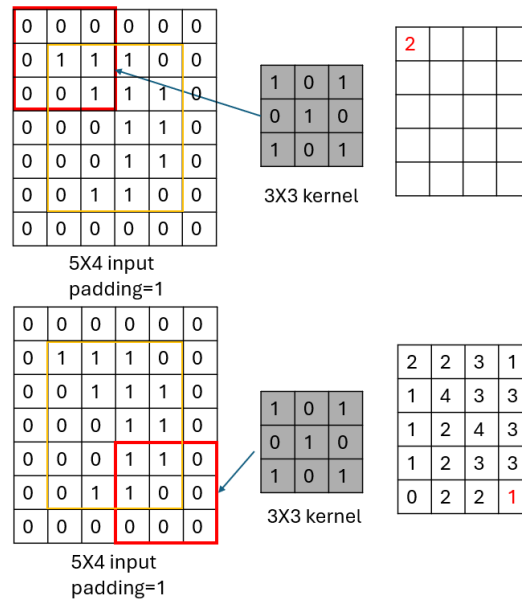


Figure2- 8 Convolution Process with Padding = 1

In a convolution operation, when the *stride* is set to 1, the sizes of *padding* and *kernel\_size* directly affect the dimensions of the input and output feature maps. Specifically, when  $K_1=K_2>1$  and the padding value is set to 0, the output feature map will have smaller dimensions than the input after convolution. The operation process and results are as follows:

4x5 matrix , 3x3 kernel , s=1 , padding=0

$$\text{new\_Height} = \frac{(5+2 \times 0 - 1 \times (3-1) - 1)}{1} + 1 = 3$$

$$\text{new\_Width} = \frac{(4+2 \times 0 - 1 \times (3-1) - 1)}{1} + 1 = 2$$

Given an input array of size 4x5, after applying the convolution, the resulting output array is reduced to a size of 2x3.

If the goal is to maintain the same output dimensions as the input, an appropriate padding value must be added to the input array. This padding can be calculated by rearranging Equation (2.2), where fixing the output dimensions equal to the input allows for determining the necessary padding =  $\frac{(k-1)}{2}$  . For example, when the kernel size is 3x3,

this formula yields a required padding of 1 to preserve the dimensions of the output feature map. The corresponding operation process and result are shown below.

$$4 \times 5 \text{ matrix}, 3 \times 3 \text{ kernel}, s=1, \text{padding}=1$$

$$\text{new\_Height} = \frac{(5 + 2 \times 1 - 1 \times (3 - 1) - 1)}{1} + 1 = 5$$

$$\text{new\_Width} = \frac{(4 + 2 \times 1 - 1 \times (3 - 1) - 1)}{1} + 1 = 4$$

From the above example, the output feature map retains the same size as the input, namely  $4 \times 5$ , after the convolution operation.

There are several types of padding methods used in convolution operations, such as replicate padding, same padding, and zero padding. In YOLO, convolution is implemented using PyTorch's Conv2d function [16], which applies zero padding. Zero padding is a technique that expands the input matrix by adding zeros around its borders before performing the convolution. Based on the specified padding value, an equal number of zeros are added to the top, bottom, left, and right sides of the input matrix. For example, given an input matrix of size  $m \times n$  if the padding value is  $P_d$ , then  $P_d$  rows of zeros are added to both the top and bottom, and  $P_d$  columns of zeros are added to the left and right, resulting in an expanded matrix of size  $(m + 2P_d) \times (n + 2P_d)$ . This method preserves effective information in the input feature map while maintaining the dimensions of the output. As illustrated in Figure 2-9, when the padding value is set to 1, a single layer of zeros is added around the input matrix, allowing the convolution kernel to operate properly near the edges without reducing the size of the output feature map. Moreover, according to Equation (2.2), the *stride* setting directly affects the height and width of the output. A stride value greater than 1 reduces the spatial dimensions of the output array. Therefore, the choice of padding and stride is critical to the convolutional network's

ability to extract features effectively, manage computational efficiency, and control the size of the resulting output.

### 2.2.3 YOLOv9-tiny Architecture

The YOLO model adopted in this study is YOLOv9-tiny, the smallest variant within the YOLOv9 series [1]. Compared to the full version of YOLOv9, YOLOv9-tiny sacrifices some accuracy but significantly reduces the number of model parameters, resulting in lower computational resource requirements. As such, YOLOv9-tiny is particularly well-suited for resource-constrained environments, such as edge computing. The overall architecture of YOLOv9-tiny and the internal structure of its modules are illustrated in Figure 2-9. The model comprises multiple modules, with detailed architecture shown in Figures 2-10 and 2-11. Different modules are color-coded according to their functional characteristics to clearly present the model's composition and operational flow.

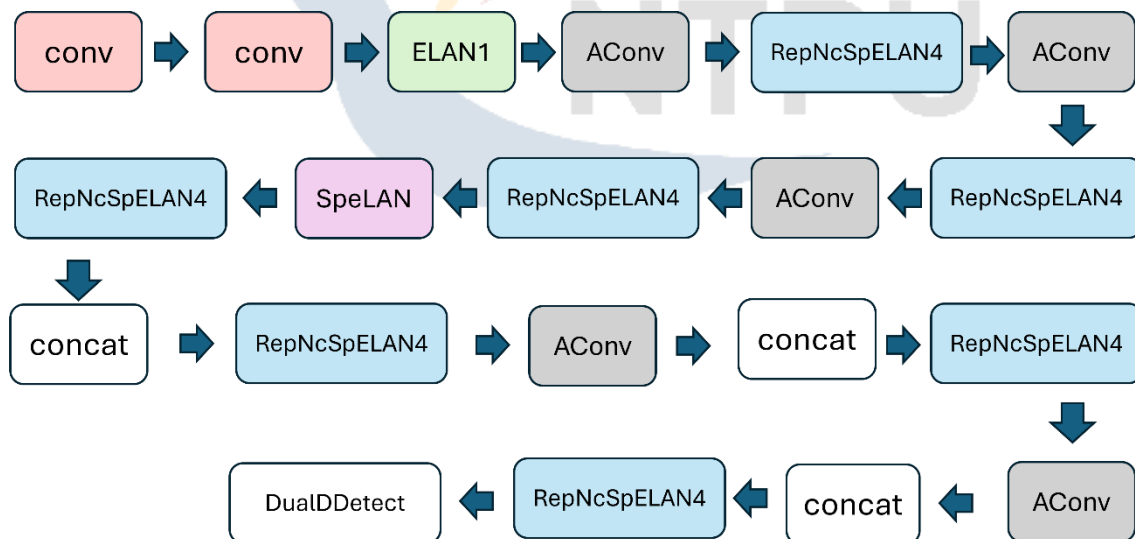


Figure2- 9 Architecture of YOLOv9-tiny

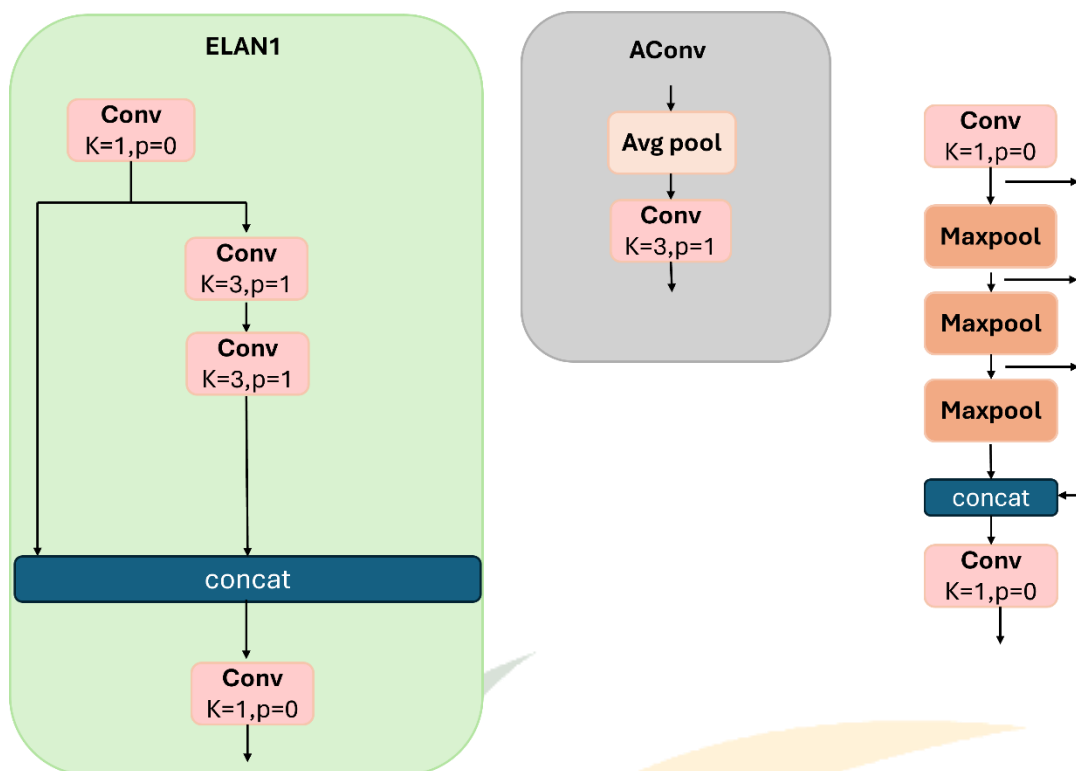


Figure2- 10 Internal Module of YOLOv9-tiny (1)

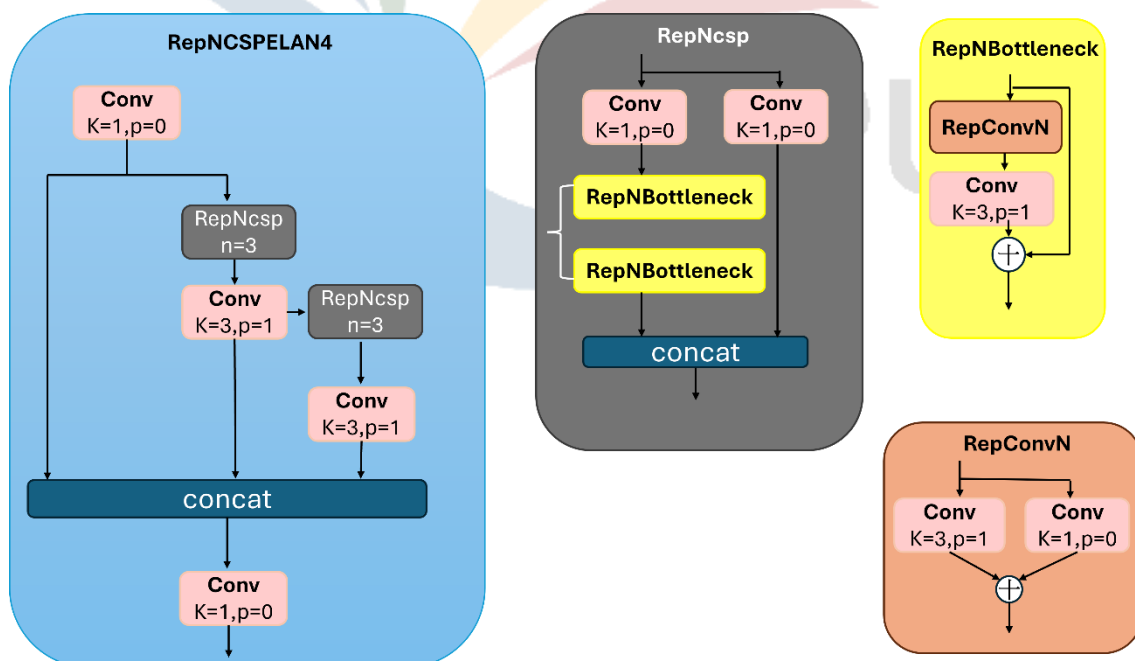


Figure2- 11 Internal Module of YOLOv9-tiny (2)



## 2.3 Overview of Vitis HLS, Vivado, and PYNQ-ZU

### 2.3.1 Vitis HLS

Vitis HLS (High-Level Synthesis) is a software tool developed by AMD that allows C or C++ code to be converted into RTL (Register Transfer Level) files written in hardware description languages (HDL) such as Verilog or VHDL. A simplified flow of this conversion process is shown in Figure 2-12. This makes hardware design more accessible by reducing the need to deeply learn HDL syntax, thereby lowering the entry barrier for hardware development. In Vitis HLS, developers can use specialized directives to allocate and optimize hardware resources. For example, as shown in Figure 2-13, certain data can be stored in LUTRAM (Look-Up Table Random Access Memory) using specific commands. Developers can also control loop unrolling and pipelining to adjust performance based on available hardware resources and system requirements. Vitis HLS also offers a complete simulation and verification environment, allowing developers to test the correctness of their design logic at the C/C++ level before final hardware deployment. This early-stage simulation helps reduce the risk of discovering errors late in the development cycle, saving significant time on debugging and redesigning. Overall, Vitis HLS not only speeds up the process from high-level algorithms to hardware implementation but also provides flexible and fine-grained resource management and optimization, making FPGA design more efficient and user-friendly.

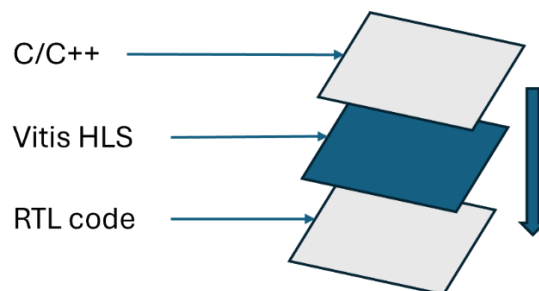


Figure2- 12 Workflow of Vitis HLS Conversion

```
#pragma HLS RESOURCE variable=local_input_part1 core=RAM_2P_LUTRAM
#pragma HLS RESOURCE variable=local_input_part2 core=RAM_2P_LUTRAM
```

Figure2- 13 Examples of Hardware Control Directives in Vitis HLS

## 2.3.2 Vivado Design Suite

Vivado Design Suite is a professional FPGA design and development tool developed by AMD. It provides a complete and unified workflow that integrates RTL coding, simulation, synthesis, hardware implementation, and deployment, significantly improving development efficiency and reducing integration issues caused by switching between different tools. Vivado supports multiple hardware description languages, such as VHDL and Verilog, giving it great flexibility and compatibility. It also supports direct import of RTL files generated by high-level synthesis tools like Vitis HLS, allowing developers to implement FPGA designs more conveniently. With its built-in IP Integrator, users can use a graphical interface to integrate and automatically connect IP cores, greatly simplifying complex system designs. In terms of design optimization, Vivado provides detailed reports on the usage of FPGA resources such as BRAM (Block RAM), LUT (Look-Up Table), and others, allowing users to clearly understand the resource allocation and make performance adjustments based on identified bottlenecks.

## 2.3.3 PYNQ-ZU

PYNQ-ZU is a high-performance FPGA development platform launched by AMD. The SoC (System on Chip) used in this board is the Zynq UltraScale+ XCZU5EG-1SFVC784 MPSoC, which integrates both FPGA fabric and ARM-based processors. The ARM section includes four Cortex-A53 cores, and the system comes with 4GB of DDR4 2400 memory on the software side. This architecture allows developers to switch flexibly

between hardware and software operations, selecting the most suitable computational method based on application needs. Since this research focuses on the FPGA part of the PYNQ-ZU, and hardware design tends to be more constrained than software design especially in terms of logic units and memory resources this study will further explain the hardware resource limitations of the PYNQ-ZU development board.

The main storage unit in FPGA is BRAM. The PYNQ-ZU board contains 144 BRAM blocks, each with a capacity of 36 Kb, totaling approximately 5.1 Mb. BRAM often becomes a bottleneck in FPGA hardware design. Another key resource is the LUT (Look-Up Table), which is used to implement combinational logic and is widely applied in digital circuit design. In addition, LUTs can be configured as LUTRAM to provide extra storage capacity beyond BRAM. The PYNQ-ZU development board includes 117,120 LUTs, of which 57,600 can be configured as LUTRAM. Besides resource limitations, hardware-level constraints must also be considered. For example, if multiple computing units attempt to access the same memory location simultaneously, data access conflicts may occur. Moreover, Vivado supports configuring up to four IP cores to write to the FPGA simultaneously on this board. The internal AXI bus, which is the high-speed communication interface between hardware and software, is also limited in number this PYNQ-ZU board provides a total of 9 AXI buses.

# Chapter 3: Hardware/Software Co-Design

## 3.1 Environment Configuration

This thesis implements the target functionality on the FPGA using C language as the basis. The Vitis HLS tool is used to convert the C code into RTL (Register-Transfer Level) code. The process begins with C Synthesis, and once synthesis is complete, Export RTL is performed to generate RTL files written in a hardware description language. The interface of this step is shown in Figure 3-1. After these steps, the generated RTL files can be imported into Vivado, AMD's design and implementation tool, for further system integration and hardware realization.

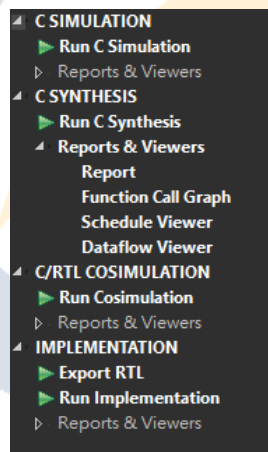


Figure3- 1 Vitis HLS Interface Layout e

In Vivado, this study uses Create Block Design to import the RTL files generated from Vitis HLS. The import process is shown in Figure 3-2. Specifically, the user first opens Settings, navigates to the IP options, selects Repository, and loads the RTL files produced by Vitis HLS. The system will then recognize the files as an IP block. Through this method, the IP block modules generated by Vitis HLS can be integrated into Vivado's

design environment and further connected with other IPs to build a complete hardware architecture.

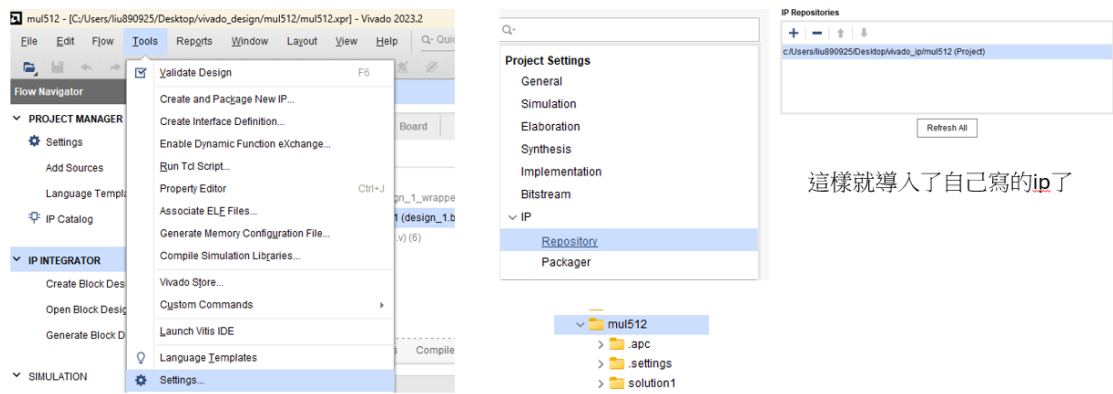


Figure3- 2 Steps for Importing IP into Vivado

When applying the IP module, a new design must first be created under Create Block Design. On the design canvas, the user right-clicks and selects Add IP, then enters the name of the function previously created in Vitis HLS to search for the corresponding IP module. By selecting the desired module, it is successfully imported into the project. The detailed steps are shown in Figure 3-3. After the IP module is added, the user can click Run Block Automation, and Vivado will automatically configure the necessary interconnections between modules. The result is shown in Figure 3-4.

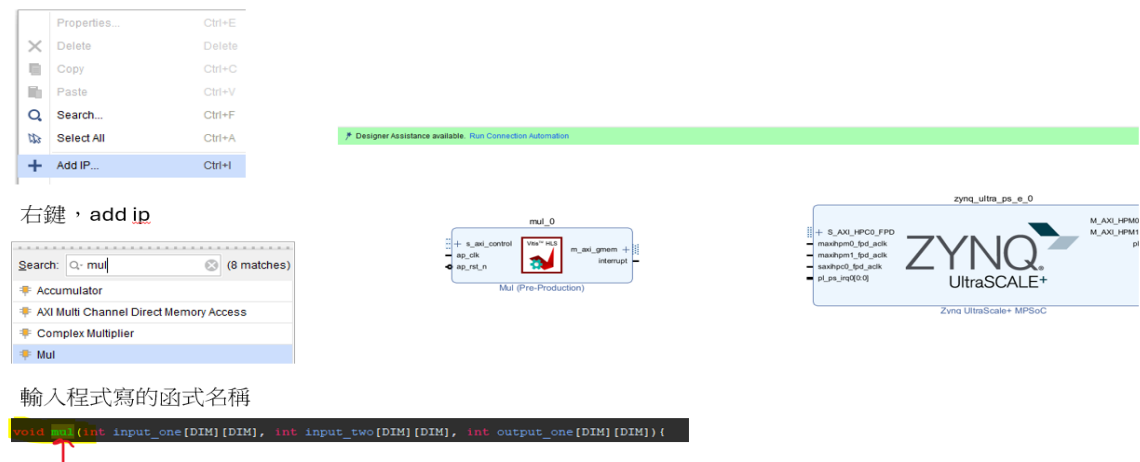


Figure3- 3 Usage of Custom IP in Vivado

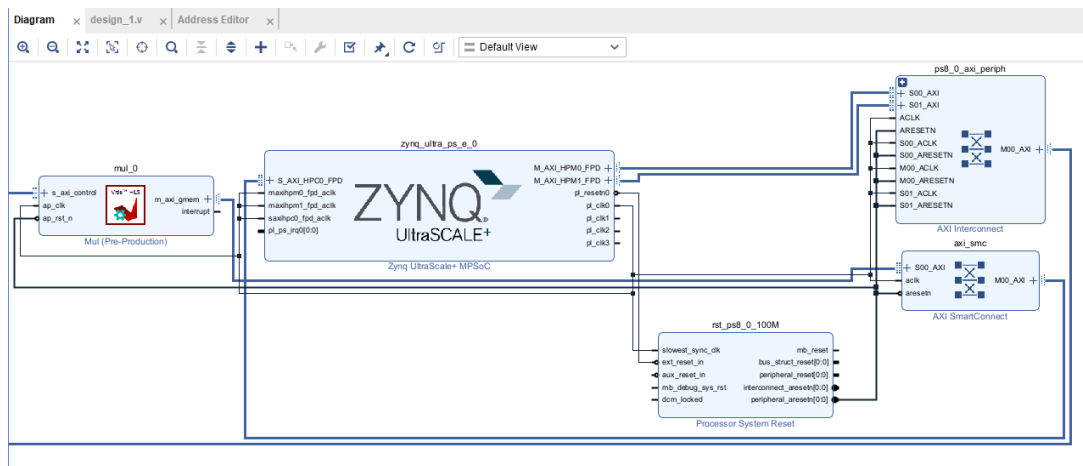
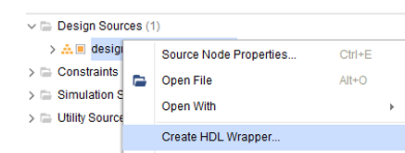


Figure3- 4 Finalized IP Connection Diagram

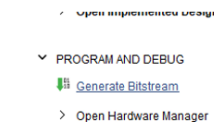
After the IP design and connections are completed, the next step is to generate the bitstream. First, right-click on the Block Design interface and select Create HDL Wrapper, which generates an HDL (Hardware Description Language) wrapper file that describes the overall hardware structure and serves as the foundation for the next steps. Then, execute Generate Bitstream. Through this process, Vivado will automatically perform synthesis, implementation, and bitstream generation, ultimately producing two output files: .bit and .hwh, as shown in Figure 3-5.



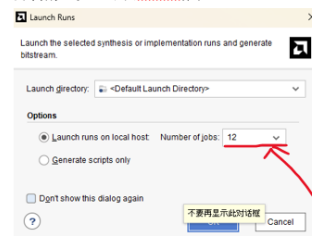
右鍵設計模塊，並點擊圖上，出現提示框直接按ok



完成後



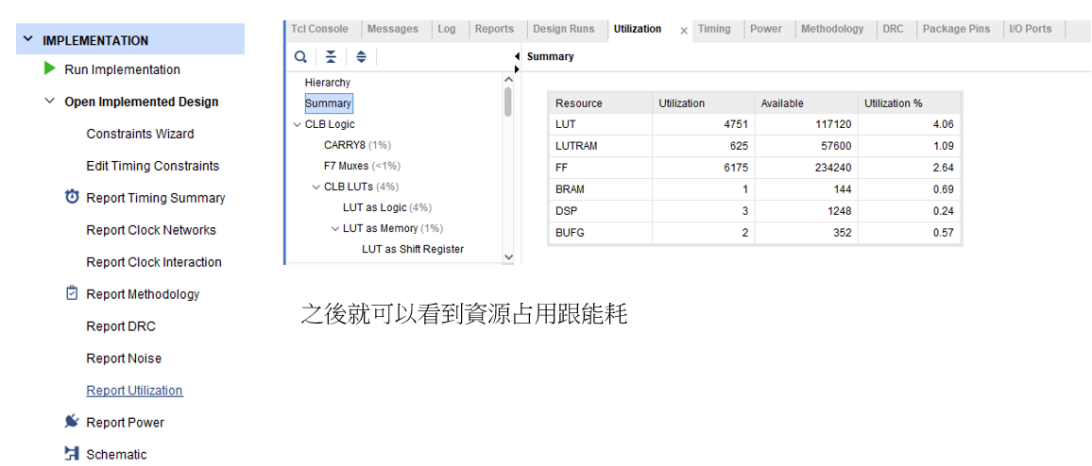
完成後就即可按generate bitstream 就會產生所需的bit 跟hwh檔



可以依照電腦性能選擇，會影響產生速度

Figure3- 5 Workflow for Generating .bit and .hwh Files

Once the .bit and .hwh files have been generated, the hardware resource utilization of the design can be reviewed using Report Utilization under the Implementation tab. This report provides detailed insights into the usage and allocation of hardware resources such as LUT (Look-Up Table), FF (Flip-Flop), DSP (Digital Signal Processor), and BRAM (Block RAM). The analysis results are shown in Figure 3-6. This report allows developers to evaluate whether the design meets the resource constraints of the FPGA and make necessary optimization adjustments to improve hardware efficiency.



之後就可以看到資源占用跟能耗

Figure3- 6 Viewing FPGA Hardware Resource Utilization

## 3.2 Analysis and Design of Parallel Processing

### 3.2.1 Convolution Mathematical Expression

Under traditional ARM and CPU architectures, convolution operations are typically executed in a serialized, step-by-step manner. In other words, for each convolution computation, the ARM or CPU must first complete the multiplication between a group of input data and its corresponding weights, then perform the accumulation of the results. Only after processing one complete set of data will the system proceed to the next. As a result, when dealing with large volumes of data, the computational efficiency is significantly constrained. Taking the YOLO model as an example a deep learning vision model that performs object detection the model usually contains many convolutional layers, each requiring many multiply-accumulate operations. Due to the sequential nature of these computations, when the parameter size becomes large, this one-by-one computation approach significantly increases the total execution time, becoming a major performance bottleneck in the system. Figure 3-7 illustrates the basic process of convolution computation. In this process, the convolution calculation is handled by a Conv compute module, whose core task is to perform the mathematical operations described in Equation (3.1).

$$Y_m[i,j] = \sum_{p=0}^{C_{\ell-1}-1} \sum_{K_1=0}^{K_1-1} \sum_{K_2=0}^{K_2-1} \Omega_{m,p}[K_1, K_2] X_p[i + K_1, j + K_2] \quad (3.1)$$

In the convolution mathematical expression (3.1),  $\ell$  denotes the current convolution layer. The parameter  $m$  ranges from 0 to  $C_{\ell-1}$ ; the parameter  $i$  ranges from 0 to  $W_{\ell-1} - K_1$ ; and the parameter  $j$  ranges from 0 to  $H_{\ell-1} - K_2$ .



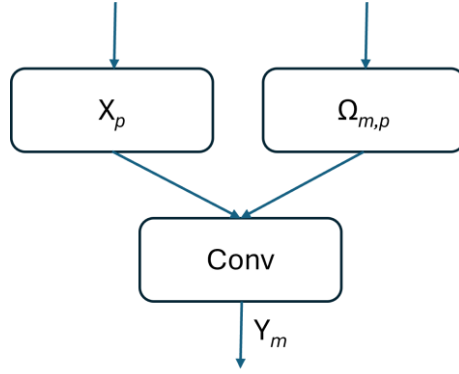


Figure3- 7 Convolution Computation Structure

In expression (3.1), each symbol corresponds to a component in the convolution operation: the input ( $X_p$ )、the weight ( $\Omega_{m,p}$ ) and the output ( $Y_m$ ). The detailed meanings and explanations of each component are described as follows. :

- i. Output : In equation (3.1), the output is defined as  $Y_m[i,j]$ , representing the result of the convolution operation at the current layer i.e., the output of the Conv operation shown in Figure 3-7. The symbol  $\ell$  (layer) indicates the current layer, and  $C_\ell$  denotes the total number of output channels at this layer. The variable  $m$  represents the current output channel and ranges from 0 to  $C_{\ell-1}$ . Since  $\ell$  refers to the current layer,  $W_\ell$  indicates the width of  $Y_m[i,j]$ . The variable  $i$  represents the horizontal position, and its range can be derived from equation (2.2). According to that equation, the value of  $W_\ell$  is influenced by  $W_{\ell-1}$ . In equation (3.1), the convolution parameters are set as dilation = 1 and stride = 1, which are the fixed settings for all convolution operations discussed in this thesis. Moreover, equation (3.1) assumes padding = 0, while cases with non-zero padding will be discussed in later chapters. Under these assumptions, the output width can be calculated as  $W_\ell = (W_{\ell-1} - K_1) + 1$ , and since the index  $i$  starts from 0, the valid range of  $i$  in equation (3.1) is  $0 \leq i \leq W_{\ell-1} - K_1$ :

$$W_{\ell} = (W_{\ell-1}) - (K_1) + 1$$

$$\Rightarrow W_{\ell} - 1 = W_{\ell-1} - K_1$$

Since  $\ell$  (layer) denotes the current convolutional layer,  $H_{\ell}$  represents the height of  $Y_m[i,j]$  at this layer. The variable  $j$  refers to the vertical direction, and its valid range can be derived from equation (2.2). According to that equation, the value of  $H_{\ell}$  is influenced by  $H_{\ell-1}$ . In equation (3.1), the parameters are set as dilation = 1 and stride = 1, which are the default settings applied to all convolution operations discussed throughout this thesis. Additionally, equation (3.1) assumes padding = 0; cases involving non-zero padding will be addressed in later chapters. Under these assumptions, the height of the output feature map is calculated as  $H_{\ell} = (H_{\ell-1} - K_2) + 1$ . Since  $j$  is defined to start from 0,  $H_{\ell-1}$  becomes the upper bound. Therefore, the valid range for  $j$  in equation (3.1) is  $0 \leq j \leq H_{\ell-1} - K_2$ :

$$H_{\ell} = (H_{\ell-1}) - (K_2) + 1$$

$$\Rightarrow H_{\ell} - 1 = H_{\ell-1} - K_2$$

Therefore, based on the above, each  $Y_m[i,j]$  corresponds to a feature map of size  $W_{\ell} \times H_{\ell}$ , and the entire output of the layer consists of  $C_{\ell}$  such feature maps, each with dimensions  $W_{\ell} \times H_{\ell}$ .

- ii. **Weight :** In Equation (3.1), the weights are defined as  $\Omega_{m,p} [K_1, K_2]$ , representing the set of weights used in the current convolution layer, as illustrated by  $\Omega_{m,p}$  in Figure 3-7.  $\Omega_{m,p} [K_1, K_2]$  are applied to perform convolution operations on the feature maps obtained from the previous layer. Here,  $K_1$  and  $K_2$  denote the horizontal and vertical indices within the kernel, respectively. These two indices define the size

of the convolution kernel, with  $K_1$  representing the kernel width and  $K_2$  representing the kernel height. The ranges of these indices are defined as  $K_1$  from 0 to  $K_1 - 1$ , and  $K_2$  from 0 to  $K_2 - 1$ . Furthermore, the variables  $m$  and  $p$  in  $\Omega_{m,p}[K_1, K_2]$  are used to distinguish between different kernels. In total, there are  $C_\ell \times C_{\ell-1}$  kernels, each of size  $K_1 \times K_2$ , where  $C_\ell$  represents the number of output channels in the current layer, and  $C_{\ell-1}$  denotes the number of input channels.

- iii. Input : In Equation (3.1), the input is defined as  $X_p[i + K_1, j + K_2]$ , representing the input data entering the current convolution layer, as illustrated by  $X_p$  in Figure 3-7. This input originates from the output of the previous layer and can thus be regarded as the output of layer  $\ell - 1$ . The symbol  $\ell$  indicates the current layer, and  $\ell - 1$  represents the preceding layer. Therefore,  $C_{\ell-1}$  denotes the total number of input channels, and the variable  $p$  identifies the index of the current input channel, ranging from 0 to  $C_{\ell-1} - 1$ . The variables  $i$  and  $j$  correspond to the horizontal and vertical coordinates of  $Y_m[i, j]$ , respectively, and their ranges have been previously discussed. When  $K_1$  and  $K_2$  are greater than 1, adding  $K_1$  and  $K_2$  to  $i$  and  $j$ , respectively, allows  $X_p$  to correctly align with the convolution kernel and ensure proper computation over the relevant region. Based on the above, Equation (3.1) describes the complete process of a convolution operation, explaining how the output  $Y_m[i, j]$  of the current layer is computed from the input data  $X_p[i + K_1, j + K_2]$  and the weights  $\Omega_{m,p}[K_1, K_2]$ . Each value of  $Y_m[i, j]$  is the result of a multiply-accumulate operation applied over the corresponding region of the input  $X_p[i + K_1, j + K_2]$  and the associated weights  $\Omega_{m,p}[K_1, K_2]$ .

$$Y_m[i, j] = \sum_{p=0}^{C_{\ell-1}-1} \Omega_{m,p}[0, 0] X_p[i, j] \quad (3.2)$$

According to the computational process described in Equation (3.1), the core of the convolution operation lies in performing element-wise multiplications within a local receptive field, followed by summing these products to produce the final output. Each result from the multiply-accumulate operation is then correctly stored in the corresponding position of  $Y_m [ i , j ]$ , thus completing the output feature map. When the kernel size is reduced to  $1 \times 1$ , meaning  $K_1=1$  and  $K_2=1$ , Equation (3.1) can be further simplified. In this case, the kernel contains only a single element, and both parameters  $K_1$  and  $K_2$  are fixed at 0. As a result, the weight  $\Omega_{m,p} [ K_1 , K_2 ]$  simplifies to  $\Omega_{m,p} [ 0 , 0 ]$ , and the corresponding input value becomes  $X_p [ i , j ]$ . This implies that the originally more complex operation, which involves traversing and processing  $K_1 \times K_2$  elements, is now reduced to a single multiplication and accumulation per output element. The corresponding equation can be simplified from (3.1) to the form shown in Equation (3.2). In Equation (3.2),  $\ell$  denotes the current convolutional layer,  $m$  ranges from 0 to  $C_{\ell-1}$ ;  $i$  ranges from 0 to  $W_{\ell-1} - 1$ , and  $j$  ranges from 0 to  $H_{\ell-1} - 1$ .

$$Y_m [i,j] = \sum_{p=0}^{C_{\ell-1}-1} \sum_{K_1=0}^{K_1-1} \sum_{K_2=0}^{K_2-1} \Omega_{m,p}[k_1, k_2] X_p' [i + K_1 - (\frac{K_1-1}{2}), j + K_2 - (\frac{K_2-1}{2})] \quad (3.3)$$

Equation (3.3) describes the convolution operation with padding applied. In the later sections of this thesis, the amount of padding added is uniformly denoted as Pd. As previously discussed, when the kernel size satisfies  $K_1, K_2 > 1$ , and  $Pd = 0$ , the output dimensions will be smaller than the input dimensions, as shown in the calculation example in Figure 2-9. To maintain consistent dimensions between the input and output, an appropriate padding value must be added, as illustrated in the computation in Figure 2-10. In Equation (3.3),  $\ell$  represents the current convolutional layer,  $m$  ranges from 0 to  $C_{\ell-1}$ ,  $i$  ranges from 0 to  $W_{\ell-1} - 1$ , and  $j$  ranges from 0 to  $H_{\ell-1} - 1$ . Compared to Equations

(3.1) and (3.2), the primary difference in Equation (3.3) lies in how the input data is represented. The input is expressed as  $X_p' [i+K_1-(\frac{K_1-1}{2}), j+K_2-(\frac{K_2-1}{2})]$ , where  $X_p'$  denotes the new input matrix obtained after padding is applied. The dimensions of this matrix are adjusted based on the size of Pd, which in turn depends on the kernel size  $K_1$  and  $K_2$ . The value of Pd can be calculated using Equation (2.2). An example of the Pd calculation for the horizontal direction is shown as follows:

$$W_{\ell-1} = W_{\ell} + 2 \times Pd - (K_1 - 1) - 1 + 1$$

$$\Rightarrow 2 \times Pd = K_1 - 1$$

$$\Rightarrow Pd = (\frac{K_1-1}{2})$$

Based on the above calculation, the padding value Pd is determined as  $(\frac{K_1-1}{2})$ , indicating that the horizontal padding value varies according to the kernel size  $K_1$ . For instance, when  $K_1=3$ ,  $Pd=1$ . In the YOLO architecture, a zero-padding strategy is adopted. When  $Pd=1$ , it means that a border of zeros is added around the input matrix  $X_p$ , effectively increasing both the  $W_{\ell-1}$  and  $H_{\ell-1}$  of the original input by 2 pixels, resulting in a new input matrix  $X_p'$ . In other words, when padding is applied to  $X_p$ , a Pd wide border is added to all sides top, bottom, left, and right. This adjustment changes the input dimensions such that the new input matrix  $X_p'$  has dimensions  $W_{\ell-1} + 2 \times Pd$ ,  $H_{\ell-1} + 2 \times Pd$ , respectively.

In order for the convolution kernel to correctly align with and fully compute the corresponding values in  $X_p'$ , the original indexing of  $X_p$  based on  $i+K_1$  and  $j+K_2$  must be adjusted to account for the added padding value Pd. The following derivation illustrates this adjustment process, using the horizontal direction as an example:

$$0 \leq i \leq W_{\ell-1} - 1, \text{an appropriate Pd value, } W_{\ell-1} = W_{\ell} - 1$$

$$\Rightarrow 0 \leq i + K_1 \leq (W_{\ell-1} - 1) + (K_1 - 1), \text{Add the same value } - (\frac{K_1-1}{2})$$

$$\begin{aligned} \Rightarrow -(\frac{K_{l-1}}{2}) &\leq i + K_{l-1} - (\frac{K_{l-1}}{2}) \leq (W_{l-1} - 1) + (K_{l-1} - 1) - (\frac{K_{l-1}}{2}) \\ \Rightarrow -(\frac{K_{l-1}}{2}) &\leq i + K_{l-1} - (\frac{K_{l-1}}{2}) \leq (W_{l-1} - 1) + (\frac{K_{l-1}}{2}) \end{aligned}$$

According to the above calculation, the final result (in the horizontal direction) shows that one padding unit of size  $Pd = (\frac{K_{l-1}}{2})$  is added to both sides of the original input. The details are as follows:

- On the left side of the horizontal dimension (i.e., the negative direction of the horizontal axis), a padding of  $(\frac{K_{l-1}}{2})$  is added. In terms of index calculation, this corresponds to  $0 + (-\frac{K_{l-1}}{2})$
- On the right side of the horizontal dimension (i.e., the positive direction of the horizontal axis), a padding of  $(\frac{K_{l-1}}{2})$  is also added. In terms of index calculation, this corresponds to  $(W_{l-1} - 1) + (\frac{K_{l-1}}{2})$

The above calculation for the horizontal direction aligns with the previously defined horizontal indexing of  $X_p'$  in the thesis. Thus, the horizontal index of  $X_p'$  can be expressed as  $i + K_1 - (\frac{K_1-1}{2})$ . Similarly, the vertical direction follows the same pattern of calculation.

For the vertical dimension of  $X_p'$ , the padding value is  $(\frac{K_2-1}{2})$ , and following the logic used in the horizontal case, the vertical index becomes  $j + K_2 - (\frac{K_2-1}{2})$ . By incorporating these adjusted indices in both horizontal and vertical directions into Equation (3.1), we arrive at Equation (3.3). This equation allows the output feature map size to remain consistent with the input feature map size when  $K_1, K_2 > 1$ .

### 3.2.2 Intuitive Parallel Processing Architecture

When executing convolution operations on the PYNQ-ZU platform, if no optimization strategies are applied and the operation is implemented in a purely serialized manner, the performance, while somewhat improved compared to using pure Python (here referring to Python code converted from C++ syntax), still cannot match the performance of PyTorch's built-in Conv2d function. The fundamental reason lies in the fact that FPGA's primary strength is its ability to perform massive parallel computations. A purely sequential design fails to fully exploit this advantage. As described in Equation (3.1), convolution involves multi-level nested loops. Without optimization, this loop-heavy structure prevents hardware resources from being fully utilized, often leading to resource limitation problems in Vitis HLS and significantly increasing latency as reported by the HLS tool. Although Vitis HLS provides the UNROLL directive to flatten loop iterations and thus enable finer-grained parallel processing, memory contention can prevent full unrolling. This leads to memory dependency warnings, which cause processing delays when multiple units attempt to access the same memory simultaneously, ultimately negating the performance gains of parallelization. Therefore, without architectural optimization, convolution on FPGA fails to leverage the hardware's parallelism and may suffer from issues such as excessive input channel allocation per Conv unit, incomplete loop unrolling, and ineffective code optimization. In real-world usage, this results in slower execution speeds compared to highly optimized deep learning libraries like Conv2d.

When convolution is executed on FPGA without effective architectural optimization, the extensive nesting and high iteration counts in the program often result in an overly large computational scale. This, in turn, restricts hardware resource allocation and

prevents full utilization of the FPGA's parallel processing capabilities. To address this issue, this research restructured the architecture by breaking down the original large-scale convolution function into smaller sub-modules. The complex convolution operation was decomposed into multiple smaller, more parallelization-friendly functions. Additionally, the DATAFLOW directive provided by Vitis HLS was employed to enable concurrent execution of these sub-functions on FPGA. This approach reduces the complexity and size of each convolution unit, frees up hardware resources, and fully leverages the FPGA's strength in parallelism ultimately enhancing the efficiency of the convolution computation and accelerating overall system performance.

FPGAs offer high degrees of parallelism and can be programmed using software-like languages to implement hardware circuits, enabling task-specific optimizations that improve performance. To address the original convolution function's excessive loop depth and iteration count which lead to unmanageable computation sizes and resource limitations this thesis proposes an intuitive architecture that reduces the scale of individual convolution tasks to enable parallel processing. The architecture, illustrated in Figure 3-8, splits a single convolution function into two separate sub-functions. Two Conv modules, enabled via the DATAFLOW directive in Vitis HLS, each process one of the sub-functions independently. The outputs,  $Y_{m,1}$  與  $Y_{m,2}$ , are then summed together in a final stage. This design enables parallel execution of computation tasks, effectively reducing the computational load of each convolution path to half of its original size, thereby enhancing FPGA's parallel processing capability and speeding up the overall convolution operation.

However, due to inherent limitations of FPGA hardware, this design presents an unavoidable drawback. To ensure that each Conv module operates independently, a separate set of input data  $X_p$  and weight parameters  $\Omega_{m,p}$  must be allocated to each module.



In other words, the same data and weights must be duplicated and loaded into each Conv unit. If only one shared set of  $X_p$  and  $\Omega_{m,p}$  資 is used across all modules, Vitis HLS will raise an error indicating that the same memory address cannot be accessed simultaneously.

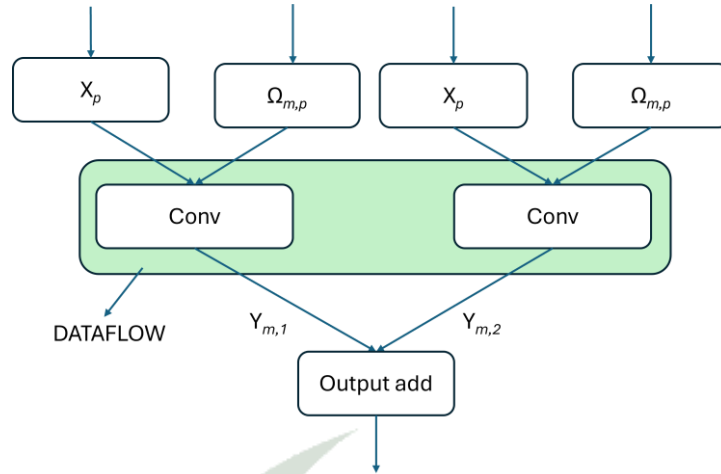


Figure3- 8 Intuitive Parallel Processing Architecture

When the convolution operation is divided into only two modules, the issue of data duplication has a relatively limited impact. However, as the convolution workload is further partitioned to increase the level of parallelism, additional Conv modules must be introduced. In such cases, each Conv module requires its own independent copy of the input feature map  $X_p$  and weight parameters  $\Omega_{m,p}$ . This rapidly exhausts the available communication channels between the Processing System (PS) and the Programmable Logic (PL), resulting in insufficient bandwidth for transferring all the necessary data. Consequently, some data may fail to be mapped to valid channels and cannot be transmitted. Moreover, the extensive data splitting and duplication in the code not only increase hardware resource consumption but also significantly complicates the program structure. This complexity reduces the readability and maintainability of overall architecture. Additionally, **resource limitation** warnings may be triggered, indicating that the available FPGA resources are insufficient to support further optimization.

Therefore, reducing redundant data input and improving resource utilization efficiency have become critical challenges in the ongoing architectural optimization process.

### 3.2.3 EPIC: Enhanced Partition for Independent Convolution

To address the issues encountered in the previous architecture, this study proposes an enhanced optimization scheme named EPIC. The primary challenge addressed by EPIC is the repeated input of  $X_p$  and  $\Omega_{m,p}$  caused by parallel convolution processing in the architecture described in Section 3.2.2. EPIC tackles this by splitting the shared parameter  $p$  which represents the number of input channels  $C_{\ell-1}$  in half, resulting in two separate partitions:  $X_{p,1}$ 、 $X_{p,2}$  and  $\Omega_{m,p,1}$ 、 $\Omega_{m,p,2}$ . In this design, the original input feature map  $X_p$  and  $\Omega_{m,p}$  are divided into two independent sub-blocks, denoted as  $X_{p,1}$ 、 $X_{p,2}$  以及  $\Omega_{m,p,1}$ 、 $\Omega_{m,p,2}$  respectively. The architecture is illustrated in the upper half of Figure 3-9. The key objective of this design is to overcome the memory access conflict encountered in the previous Figure 3-8 architecture, where sharing a single copy of  $X_p$  and  $\Omega_{m,p}$  across multiple Conv modules led to access violations in Vitis HLS due to simultaneous reads from the same memory address.

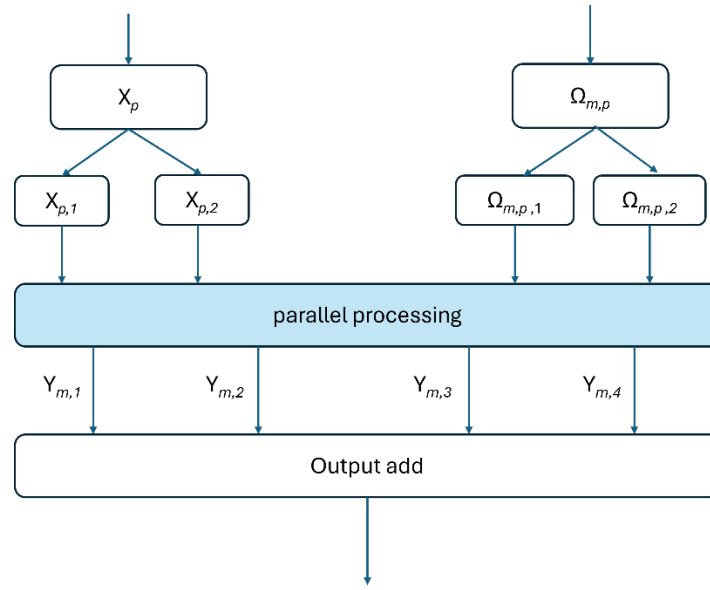


Figure3- 9 EPIC Architecture

The Parallel Processing architecture shown in Figure 3-10 represents the core region of the EPIC design introduced in Figure 3-9, and its primary function is to enable parallel execution of convolution operations. In this design, the number of Conv modules increased from two to four, thereby reducing the computational load of each Conv module to one-fourth of the original. By increasing the number of Conv modules to four, the number of input channels assigned to each module is significantly reduced. In the original unpartitioned design, a single Conv module had to process too many input channels, preventing the use of the UNROLL directive to parallelize computations across all channels. With the input channels evenly divided into four groups, each Conv module handles only a subset of the channels, making it feasible to apply the UNROLL directive to the inner loop for parallel processing. As each Conv module now processes a smaller, independent portion of the input, there is no longer a need to further split the convolution into sub-functions or use the DATAFLOW directive. Each Conv module independently computes its partial result  $Y_{m,1}$ ,  $Y_{m,2}$ ,  $Y_{m,3}$ ,  $Y_{m,4}$  which are finally summed together to produce the final convolution output.

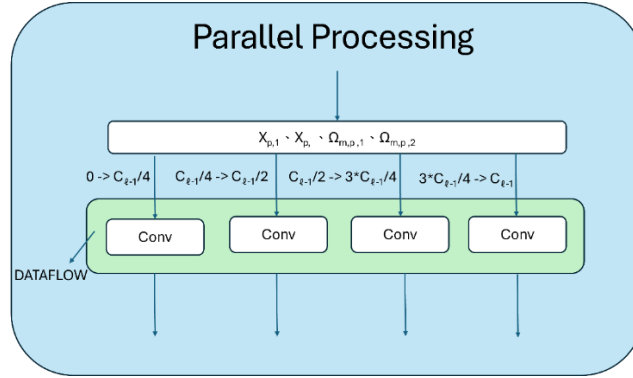


Figure3- 10 Parallel Processing Module

Figure 3-10 illustrates the overall architecture of Parallel Processing, which achieves parallel computation by dividing the number of input channels  $C_{\ell-1}$  in Equation (3.3) into four segments. This partitioning allows a single convolution function to be split evenly into four sub-convolution functions, each processed independently. The partitioning method is defined as follows:

- Segment 1:  $0 \rightarrow C_{\ell-1}/4$
- Segment 2:  $C_{\ell-1}/4 \rightarrow C_{\ell-1}/2$
- Segment 3:  $C_{\ell-1}/2 \rightarrow 3 \times C_{\ell-1}/4$
- Segment 4:  $3 \times C_{\ell-1}/4 \rightarrow C_{\ell-1}$

For example, when  $C_{\ell-1}=48$ , meaning there are 48 input channels and  $K_1=K_2=1$ , the 48 channels in Equation (3.3) are divided among four Conv modules. The use of the UNROLL directive to optimize the loop in this case is as follows:

- The first Conv module is responsible for processing input channels 1 through 12 (i.e., handling the computation for Equation 3.3 where  $p=0$  to  $p=11$ ). The UNROLL directive is applied to unroll the loop from  $p=0$  to  $p=11$  for parallel execution.

- The second Conv module is responsible for processing input channels 13 through 24 (i.e., handling the computation for Equation 3.3 where  $p=12$  to  $p=23$ ). The UNROLL directive is applied to unroll the loop from  $p=12$  to  $p=23$  for parallel execution.
- The third Conv module is responsible for processing input channels 25 through 36 (i.e., handling the computation for Equation 3.3 where  $p=24$  to  $p=35$ ). The UNROLL directive is applied to unroll the loop from  $p=24$  to  $p=35$  for parallel execution
- The fourth Conv module is responsible for processing input channels 37 through 48 (i.e., handling the computation for Equation 3.3 where  $p=36$  to  $p=47$ ). The UNROLL directive is applied to unroll the loop from  $p=36$  to  $p=47$  for parallel execution

EPIC architecture integrates multiple internal optimization strategies, effectively addressing the challenges faced in previous designs and achieving superior convolution performance compared to using PyTorch's Conv2d function on an ARM-based processor. This result demonstrates the potential of the EPIC architecture in accelerating convolution operations and fulfills the primary objective of this study. By dividing  $X_p$  and  $\Omega_{m,p}$  into two parts ( $X_{p,1} \setminus X_{p,2} \setminus \Omega_{m,p,1} \setminus \Omega_{m,p,2}$ ), EPIC resolves issues encountered when applying the DATAFLOW directive for parallel execution. This data partitioning strategy reduces the size of each convolution module to one-fourth of the original, thereby addressing resource limitations caused by excessively large loop structures. Furthermore, the reduced module size enables the use of the UNROLL directive, further enhancing parallelism. Through these optimizations, the EPIC architecture effectively leverages the parallel computing capabilities of FPGA hardware, achieving better convolution performance than traditional ARM-based Conv2d implementations.

## 3.3 Shared Convolutional IP Design

### 3.3.1 Reasons for Sharing Convolution IP

In the YOLOv9 architecture, convolution is the most fundamental and frequently executed operation. As previously mentioned in this thesis, YOLOv9 performs convolution using PyTorch's built-in Conv2d function, whose computation is influenced by several parameters as shown in Equation (3.3), including  $C_{\ell-1} \cdot C_{\ell} \cdot H_{\ell-1} \cdot W_{\ell-1}$ . However, due to the significant differences between FPGA and software execution environments and the resource limitations inherent in hardware it is difficult to implement all possible combinations of convolution parameters within a single hardware function.

The key challenge lies in the loop structures involved in convolution operations, which are governed by Equation (3.3). In FPGA-based implementations, the ranges of loop variables (such as  $p$ ,  $q$ ,  $r$ ) must be predetermined before circuit layout and resource allocation. Different convolution configurations (e.g., variations in  $C_{\ell-1}$  and  $C_{\ell}$ ) directly affect these loop bounds. To address this constraint, it is necessary not only to optimize the data structures for each convolution IP (Intellectual Property) but also to adopt IP-sharing strategies to conserve hardware resources. This approach enables the FPGA to support a wider variety of convolution parameter configurations and allows YOLOv9 to better leverage the FPGA's computational capabilities, ultimately improving performance.

### 3.3.2 Shared Convolution IP

IP, also known as IP Core, functions like a hardware-based library, where each IP represents an independent and complete functional module. The PYNQ-ZU platform used in this thesis supports up to four different IPs being connected within the Vivado development environment. By integrating multiple IPs in Vivado, convolution layers with

different parameter configurations can be computed independently. Without further optimization, one EPIC corresponds to one IP, and each EPIC is designed and accelerated for a single, specific convolution configuration. Therefore, the PYNQ-ZU platform can compute a maximum of four convolution configurations. To improve hardware resource utilization and overall speed, this study implements a design where multiple IPs are packaged into a single FPGA and introduces a shared-IP approach. This method expands the coverage of each IP, enabling it to support more types of convolution configurations.

Under hardware resource constraints, this thesis proposes an optimization method to improve FPGA support for various convolution parameters. The method involves classifying convolution configurations based on frequently used parameters such as  $C_{\ell-1}$ 、 $C_{\ell}$ 、 $H_{\ell-1}$ 、 $W_{\ell-1}$  and selecting the most representative combinations to create shared templates, referred to as TIPs (Template IPs). These TIPs are written into the same FPGA via Vivado, allowing multiple TIPs to be used simultaneously. Furthermore, based on the EPIC design, each TIP supports multiple convolution operations to reduce the need for one IP per parameter set. When convolution layers share parameters or have related ones, this study allows them to use the same TIP to optimize resource usage. This approach increases the number of supported convolution configurations within the FPGA, improves resource efficiency, and enhances overall YOLOv9 performance.

In YOLOv9, convolution layers often vary in their parameter configurations. For instance, two layers may have different  $C_{\ell-1}$  and  $C_{\ell}$  values while sharing the same input dimensions  $W_{\ell-1}$  and  $H_{\ell-1}$ . Based on this observation, this thesis groups layers with identical  $W_{\ell-1}$  and  $H_{\ell-1}$  values together even if their  $C_{\ell-1}$  and  $C_{\ell}$  values differ. Within each group, the fixed part is  $W_{\ell-1}$  and  $H_{\ell-1}$ , while the variable part is  $C_{\ell-1}$  and  $C_{\ell}$ . The shared data within the EPIC architecture namely  $X_{p,1}$ ,  $X_{p,2}$ ,  $\Omega_{m,p,1}$ ,  $\Omega_{m,p,2}$  is reused by different convolution layers within the same group. However, to ensure efficient use of resources,

configurations with excessively large  $C_{\ell-1}$  or  $C_{\ell}$  values are excluded. The maximum values within each group are then selected as the upper bounds to define the TIP, ensuring that the storage capacity within the TIP is sufficient for all target operations. Finally, the number of Parallel Processing modules within each TIP is expanded accordingly to meet the computing needs of different convolution configurations, thus realizing shared computation.

For convolution computations using the same TIP across varying configurations, the EPIC architecture must be extended with additional Parallel Processing modules to match each computation's complexity. This study assigns convolution tasks to appropriate modules based on their computational loads. This prevents small-scale convolution operations from being handled by oversized processing modules, which could otherwise lead to unnecessary overhead. Through this scheduling strategy, the original EPIC architecture initially equipped with single Parallel Processing modules expanded to include three modules with different processing scales, as shown in Figure 3-11. This design significantly broadens the FPGA's ability to handle a diverse range of convolution operations, further improving the overall computational performance of the YOLOv9 system.

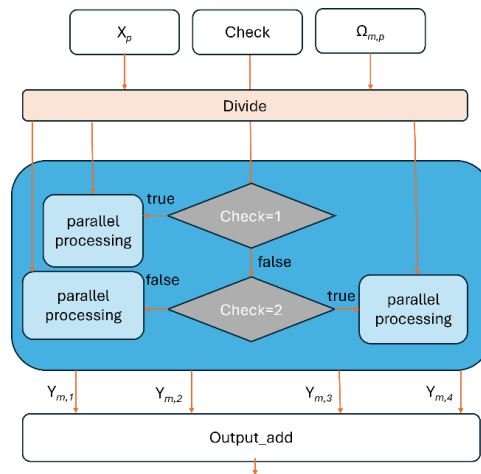


Figure3- 11 Shared Convolution Architecture



The Parallel Processing modules shown in Figure 3-11 are consistent with the design described earlier in Figure 3-10. In Figure 3-11, the Check module is used to determine which Parallel Processing module corresponds to the current convolution operation. Additionally, the Divide module, illustrated in Figure 3-12, is responsible for splitting the input  $X_p$  into  $X_{p,1} \cdot X_{p,2}$ , as well as dividing the weights  $\Omega_{m,p}$  into  $\Omega_{m,p,1} \cdot \Omega_{m,p,2}$ . The details of this partition have been thoroughly discussed in previous sections. In this study, the shared resources mainly refer to the storage modules within the EPIC architecture, which are derived from the outputs of the Divide module namely  $X_{p,1}$ 、 $X_{p,2}$ 、 $\Omega_{m,p,1}$  及  $\Omega_{m,p,2}$ . The original EPIC design, which included only a single Parallel Processing module, has been extended to incorporate three independent modules to accommodate convolution operations of different computational scales, thereby maximizing the parallel computing capabilities of the hardware.

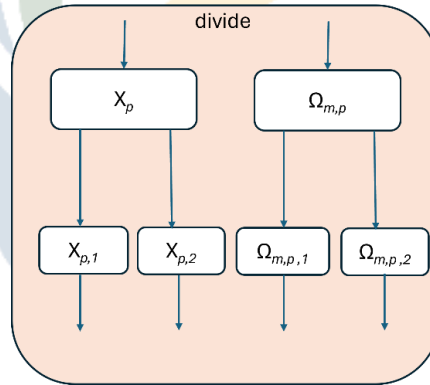


Figure3- 12 Input and Weight Division Module

### 3.3.3 Analysis of Resource Bottlenecks

To identify the bottleneck of EPIC architecture, this study adopts the most straightforward method implementing EPIC architecture without any optimization reduction into an IP core and then integrating multiple EPIC-based IPs in Vivado for resource analysis. In this unoptimized setup, the computational precision of the convolution operations is consistent with that used in PyTorch's Conv2d function, with

all data processed in float32 format. By directly deploying multiple unoptimized IPs onto the same FPGA, the hardware resource utilization can be clearly observed. This approach allows for an intuitive assessment of resource bottlenecks through Vivado's hardware utilization reports, providing a foundation for targeted optimization in future designs.

Combine three convolution IP with different parameter configurations and were deployed on the FPGA, all using float32 precision and without any IP sharing or optimization. The overall hardware resource usage after deployment is shown in Figure 3-13. Analysis of the utilization report reveals that under these conditions, BRAM usage reaches 86.11%, slightly higher than LUT usage at 70.72%, and significantly higher than LUTRAM (29%), FF (44%), and DSP (55%). These results indicate that BRAM usage is the primary bottleneck limiting the scalability of the EPIC architecture on the FPGA. BRAM, a type of embedded memory block in the FPGA, plays a crucial role in storing the various data required for computation.

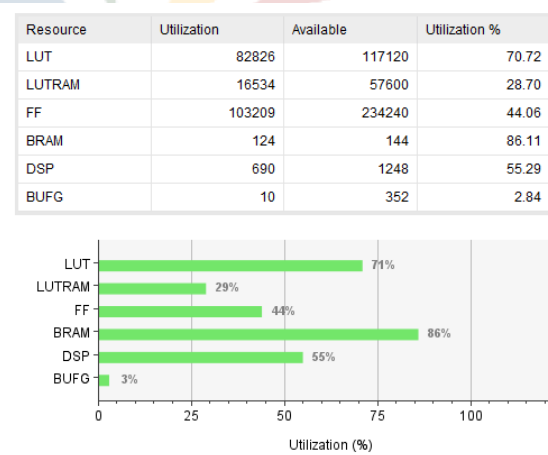


Figure3- 13 Resource Utilization of Multiple Convolution IP (Float32)

This study conducted an in-depth analysis of BRAM usage and found that the primary factor contributing to the high BRAM utilization is the storage of  $X_{p,1}$ 、 $X_{p,2}$  and  $\Omega_{m,p,1}$ 、 $\Omega_{m,p,2}$  data in float32 format. Each piece of data occupies 32 bits of memory space.

When storing high-dimensional feature maps in convolutional neural networks, the consumption of BRAM increases significantly. This issue is especially critical in YOLO models, where each feature map tends to be large, further raising BRAM storage demands and increasing the risk of memory shortage. Such constraints can negatively impact the feasibility and performance of FPGA-based designs. To address BRAM shortages, this thesis proposes in Section 3.3.2 a shared IP method, where convolution operations with the same input dimensions (identical  $W_{\ell-1}$  and  $H_{\ell-1}$ ) but different channel numbers ( $C_{\ell-1}$  and  $C_{\ell}$ ) share resources. This strategy, illustrated in Figure 3-11, involves creating a Template IP (TIP) that allows multiple convolution operations to share the BRAM-heavy components specifically,  $X_{p,1}$ 、 $X_{p,2}$  and  $\Omega_{m,p,1}$ 、 $\Omega_{m,p,2}$ . The decision to implement TIP sharing is based on the analysis of Figure 3-13, which identified BRAM as the major bottleneck of the EPIC design. This throttling strategy effectively reduces BRAM consumption and makes it possible for the FPGA to support more convolution operations. In the following sections, this thesis further investigates and optimizes the high BRAM usage caused by using float32 precision. To reduce the bit-width of each data entry while preserving model performance, we adopt the ap\_fixed data type provided by Vitis HLS in place of float32. This significantly lowers BRAM consumption and improves overall hardware efficiency. In addition, by introducing open-source support for LUTRAM, we further expand the available storage capacity on the FPGA.

### 3.3.4 Optimization Strategies for Resource Bottlenecks

This thesis adopts the built-in ap\_fixed (Arbitrary Precision Fixed Point Types) directive in Vitis HLS to replace the original float32 format. The reason for not using float16 or int8 precision formats lies primarily in Vitis HLS's lack of support for float16. Although int8 has potential to reduce resource usage, it requires specific quantization and

dequantization processes to represent fractional precision. However, Vitis HLS does not natively support such workflows. The `ap_fixed` directive allows developers to customize the total bit-width and number of fractional bits based on actual design needs. This enables precise control over numeric precision and helps minimize resource consumption in hardware implementations. This flexible data representation method supports optimal and balanced hardware resource utilization while maintaining the required model performance.

In Vitis HLS, `ap_fixed<a, b>` is a signed fixed-point data type where:

- Total bit-width (a): Defines the total number of bits used to represent a value. According to AMD's official documentation, the maximum total bit-width can be set to 1024 bits.
- Integer bit-width (b): Indicates the number of bits used for the integer part of the value, including 1 sign bit to determine the sign. This value must be smaller than the total bit-width.
- Fractional bit-width: Determined by  $a - b$ , i.e., the total bit-width minus the integer bit-width.

In numerical operations, `ap_fixed` uses the standard two's complement representation:

- When the most significant bit (sign bit) is 0, the value is positive and can be directly added using binary arithmetic.
- When the sign bit is 1, the value is negative and must be converted using two's complement to obtain the correct negative value.

For example, in Figure 3-14, both the left and right examples use `ap_fixed<3,2>`, which means a total bit-width of 3 bits, an integer bit-width of 2 bits (including 1 sign bit), and a fractional bit-width of  $3 - 2 = 1$  bit.

- In the left example, the sign bit is 0, indicating a positive number. The value is calculated as  $0 + 1/2 = 0.5$ .
- In the right example, the sign bit is 1, indicating a negative number. The value is computed using two's complement as  $-(1 + 1/2) = -1.5$

This highly flexible and customizable data representation allows developers to precisely control data precision, dynamic range, and resource consumption based on application requirements, thereby achieving an optimal balance between hardware resource utilization and performance while meeting system requirements.

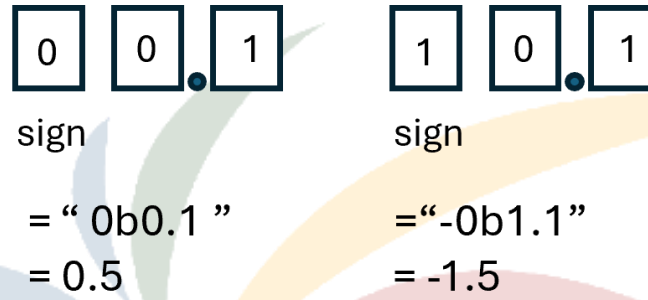


Figure3- 14 Example of `ap_fixed<3,2>`

Through testing and analysis, this study replaced all originally float32-stored data with the precision format of `ap_fixed<18,6>`. This configuration indicates a total bit width of 18 bits, with 6 bits allocated to the integer part (including 1 sign bit for determining the sign), and the remaining 12 bits ( $18 - 6$ ) allocated to the fractional part. Experimental results show that under this precision setting, the recognition accuracy experiences only a slight degradation, remaining within an acceptable range. This demonstrates a well-balanced trade-off between numerical precision and hardware resource utilization.

The `ap_fixed<a, b>` data type in Vitis HLS offers high flexibility in adjusting bit widths, allowing independent precision configurations for each storage block. For instance,  $X_{p,1}$  and  $X_{p,2}$  can be considered one group, while  $\Omega_{m,p,1}$  and  $\Omega_{m,p,2}$  form another.

Each group can have its integer and fractional bit configurations tailored based on their respective storage content and computational needs. This fine-grained precision management strategy further reduces BRAM consumption, achieving a better balance between recognition performance and hardware resource efficiency.

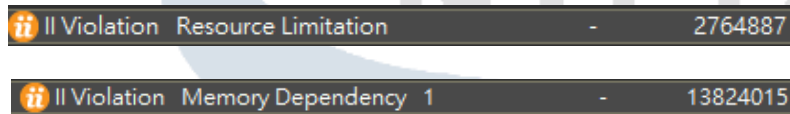
This study continues to analyze the resource utilization shown in Figure 3-13, which indicates that the LUTRAM usage is only 29%, suggesting there is still a large amount of unused LUTRAM available. LUTRAM, formed from the internal LUT configurations of the FPGA, is a small-capacity type of RAM. While it does not offer as much capacity as BRAM, it provides advantages in flexibility and is well-suited for storing small datasets. Based on this analysis, part of the data originally stored in BRAM including  $X_{p,1}$ 、 $X_{p,2}$ 、 $\Omega_{m,p,1}$ 、 $\Omega_{m,p,2}$  is transferred to LUTRAM using directives provided by Vitis HLS. This approach helps alleviate the pressure on BRAM while increasing the effective utilization of LUTRAM. Such a resource allocation strategy optimizes overall hardware resource usage and offers greater flexibility for future expansion, achieving a balanced use of memory resources.

# Chapter 4: Research Results

## 4.1 Single-Layer Convolution Acceleration Performance

### 4.1.1 DATAFLOW Architecture Performance

In the initial experiment, Equation (3.3) was adopted as the mathematical foundation for implementing convolution operations in Vitis HLS. The kernel sizes were set to  $K_1 = K_2 = 1$ , and the convolution was implemented in C++ within the Vitis HLS environment. At first, the code was synthesized without any optimization directives, resulting in a “Resource Limitation” warning in the Vitis HLS synthesis report, as shown in the upper part of Figure 4-1. The reported latency reached as high as 2,764,887 clock cycles. Later, the UNROLL directive was applied in an attempt to reduce latency by parallelizing loop operations. However, this caused a memory dependency issue, which further increased the reported latency to 13,824,015 cycles, as shown in the lower part of Figure 4-1.



ii	ll Violation	Resource Limitation	-	2764887
ii	ll Violation	Memory Dependency 1	-	13824015

Figure4- 1 Pre-Optimization Architectural Issues

To further leverage the parallel processing capabilities of the FPGA, this study first adopted the DATAFLOW directive, which allows multiple functions to execute concurrently. A convolution layer from the YOLOv9-tiny model one among the 221 total convolution layers was selected as the test case. The corresponding convolution function was split into two sub-functions, and the DATAFLOW directive was used to enable their parallel execution. The goal was to verify whether DATAFLOW could effectively improve runtime performance. As shown in Figure 4-2, the unoptimized convolution took

about 114 ms, while the version with only the DATAFLOW directive applied reduced the execution time to 65 ms. This result demonstrates that the optimized version took only 57% of the original runtime, validating the effectiveness of DATAFLOW and supporting its further use in subsequent designs.

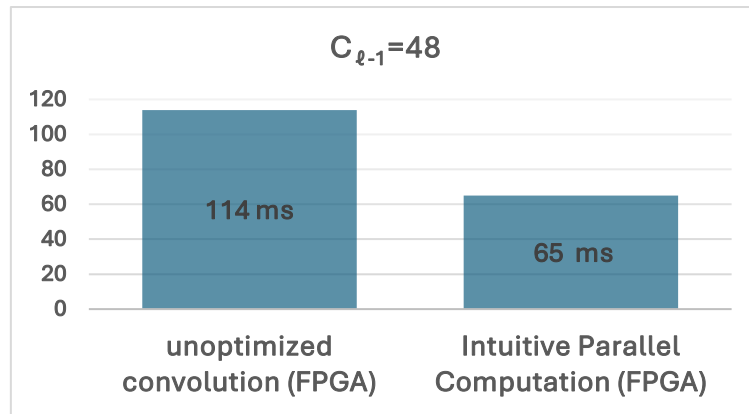


Figure4- 2 Results of DATAFLOW Optimization

### 4.1.2 EPIC Architecture Performance

Building on the performance improvement achieved using the DATAFLOW directive, this research further proposed the EPIC architecture to achieve even higher efficiency in convolution operations. Figure 4-3 compares the execution times between Python's Conv2d function and the proposed EPIC architecture under identical convolution layer conditions. The Conv2d function is a highly optimized convolution operation in Python, which, when executed on the PS side of the PYNQ-ZU platform, can utilize four Cortex-A53 processor cores for parallel computation, offering notable performance. The results show that Conv2d took approximately 5.9 ms to complete the convolution, while the EPIC architecture achieved the same task in just 2.5 ms. This significant acceleration confirms the effectiveness of deploying the EPIC architecture on FPGA and demonstrates the performance advantage of FPGAs over traditional methods in convolution computations.



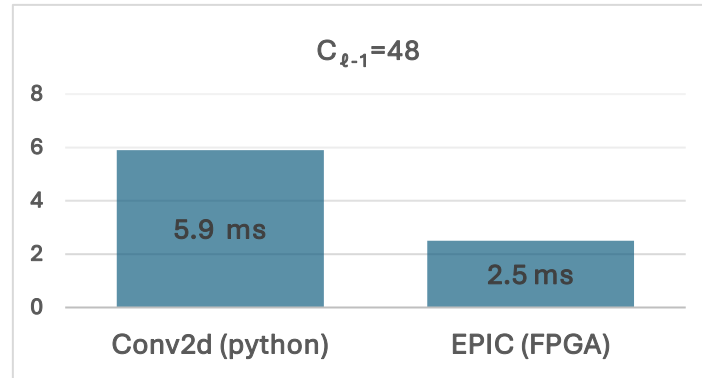


Figure4- 3 Comparison of EPIC and Conv2d Performance

Figure 4-4 compares the hardware utilization between the EPIC (float32) architecture and the previously described baseline implementation. The results indicate that all types of hardware resources show increased utilization after applying EPIC. This aligns with the primary objective of this study to fully exploit the computational resources within the FPGA to accelerate convolution operations and achieve superior performance.

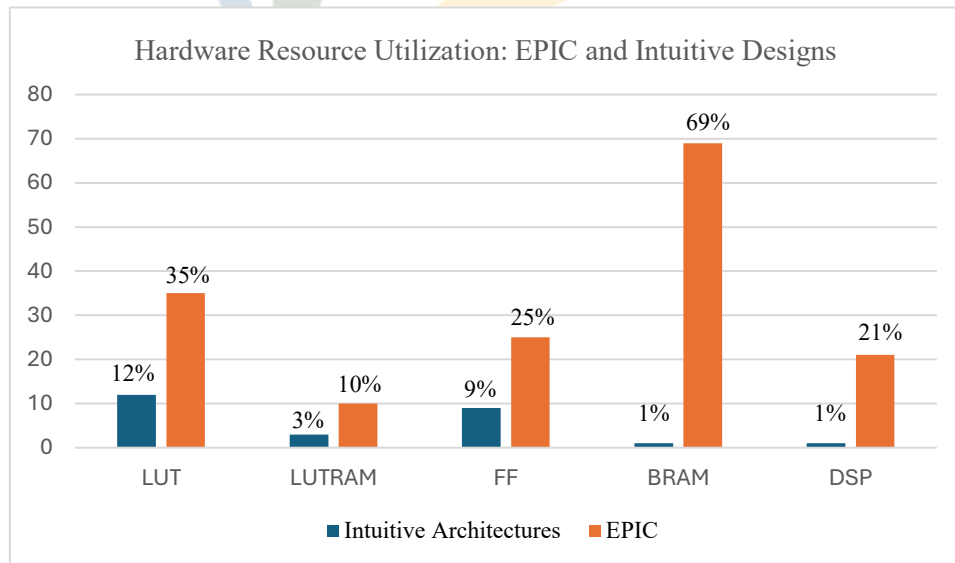


Figure4- 4 Hardware Resource Utilization: EPIC and Intuitive Designs

## 4.2 Impact of Convolution Layer Sharing on Resource Utilization

### 4.2.1 Resource Savings Achieved Using ap\_fixed<a, b>

Figure 4-5 shows a comparison of hardware resource usage for the same convolution layer in YOLOv9-tiny on FPGA when using float32 and ap\_fixed for data storage. When using float32, BRAM utilization reached as high as 69%. By optimizing the data format with the ap\_fixed directive specifically using a precision of ap\_fixed<18,6> BRAM usage significantly decreased to 36%, as seen in the figure. This result indicates that the high BRAM consumption was primarily due to the use of float32 precision. In addition to the BRAM savings, other hardware resources also saw noticeable reductions with ap\_fixed<18,6> compared to float32: LUT usage dropped from 35% to 20%, LUTRAM from 10% to 6%, FF from 25% to 13%, and DSP from 21% to 17%. These experimental results clearly demonstrate that converting data precision from float32 to ap\_fixed<18,6> can significantly reduce overall hardware resource consumption particularly memory usage, thereby improving both resource efficiency and computational performance of the FPGA design.

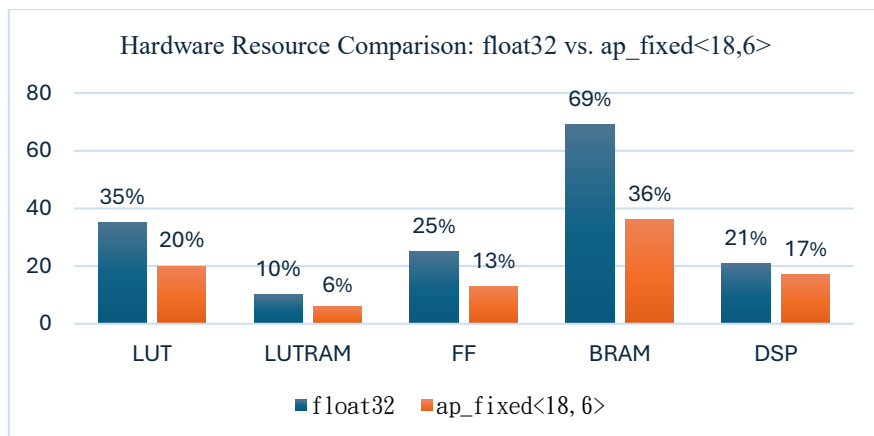


Figure4- 5 Hardware Resource Comparison: float32 vs. ap\_fixed<18,6>

## 4.2.2 Resource Optimization and TIP Sharing Analysis

In Section 3.3.2, this thesis proposes a convolution layer sharing and classification method, based on the results summarized in Figure 4-6. The figure shows the distribution of convolution layers with different parameters when  $K_1=K_2=1$ . From the data in the figure, the convolution layers can be grouped into three categories based on the combinations of  $H_{\ell-1}$  and  $W_{\ell-1}$ . Within each group, convolution layers that cause BRAM bottlenecks are removed, and three representative groups remain, as highlighted in Figure 4-6—resulting in a total of 9 convolution layers. These are categorized by input size into: large group ( $80 \times 60$ ), medium group ( $40 \times 30$ ), and small group ( $20 \times 15$ ), each containing three convolution layers with different parameter configurations, totaling 9 different convolution layers.

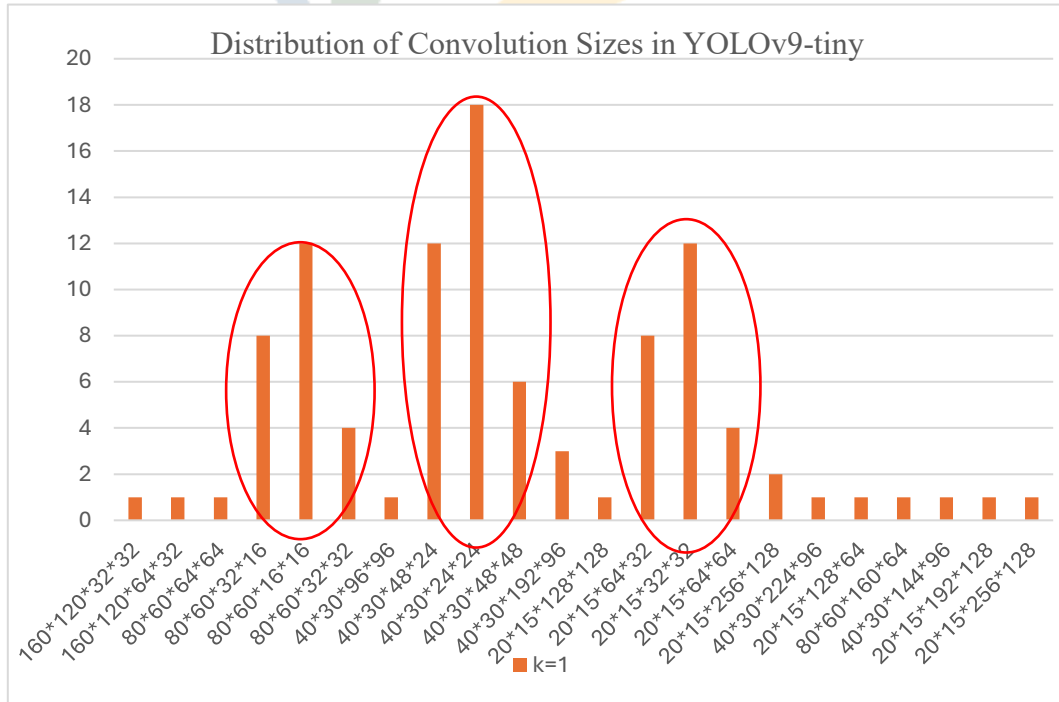


Figure4- 6 Distribution of Convolution Sizes in YOLOv9-tiny

Before optimizing with  $ap\_fixed\langle a,b \rangle$ , data was stored using float32 only. Under this setting, FPGA resources were only sufficient to store one convolution layer from the

medium group and one from the small group (2 layers total). The hardware resource usage in this case is shown in Figure 4-7. After applying the `ap_fixed<18,6>` directive, the FPGA could store one convolution layer from each of the large, medium, and small groups (3 layers total), enabling the computation of one additional convolution layer compared to float32; results are shown in Figure 4-8. Further optimization was then carried out by allocating part of the convolution data to the unused space in LUTRAM, as shown in Figure 4-9. With this method, one convolution layer from the large group and all three layers from the medium group could be stored enabling four convolution layers to be computed. Specifically, two smaller convolution layers from the medium group were stored in LUTRAM, and the remaining two layers in BRAM. The result shows a noticeable increase in LUTRAM usage in Figure 4-9 compared to Figures 4-7 and 4-8, and the increase in computable convolution layers confirms the effectiveness of this storage optimization strategy.

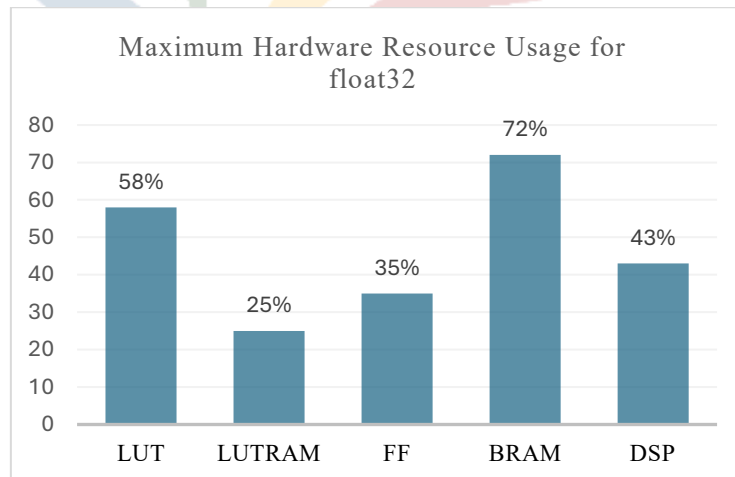


Figure4- 7 Maximum Hardware Resource Usage for float32

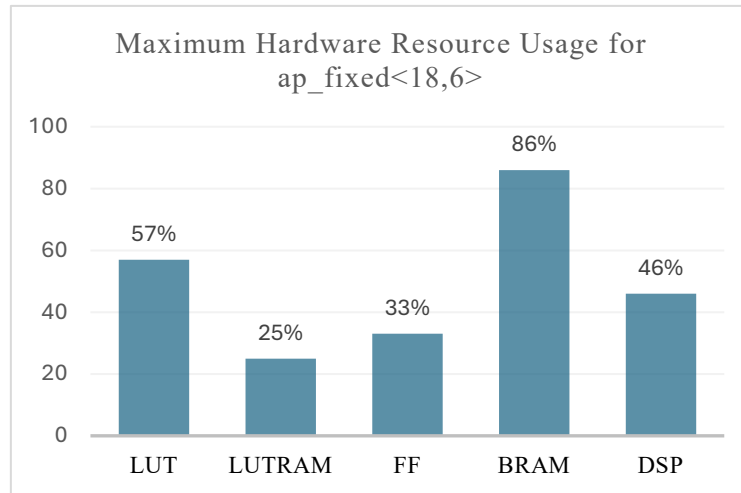


Figure4- 8 Maximum Hardware Resource Usage for ap\_fixed<18,6>

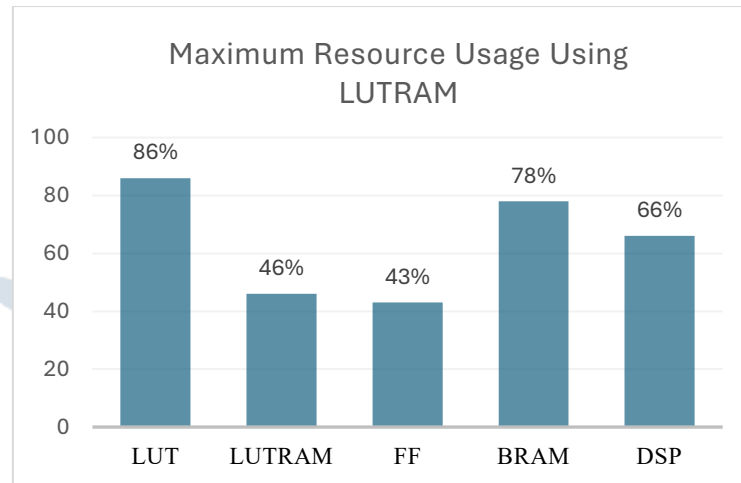


Figure4- 9 Maximum Resource Usage Using LUTRAM

The results from Figures 4-7 to 4-9 are based solely on placing multiple independent convolution IPs on a single FPGA using Vivado. While these methods improve hardware usage to some extent, they still fall short of the goal of supporting all 9 convolution layers identified in Figure 4-6. To achieve this, the concept of Template IP (TIP) is introduced. Based on the outcomes of the previous experiments, TIPs are refined with improved data precision and storage strategies. Figure 4-10 presents the BRAM usage of different TIPs using ap\_fixed<18,6>, with all data stored in BRAM. Figure 4-11 shows the LUTRAM usage when storing the same TIPs in LUTRAM instead. From Figure 4-11, it is evident

that the TIP for the large group requires 104% of available LUTRAM—causing Vivado to return an error and terminate synthesis. The estimated LUTRAM usage was 59,912 units, while only 57,600 units were available on the FPGA. Therefore, the large group TIP must be stored in BRAM, and the remaining two TIPs (for medium and small groups) are stored in LUTRAM due to insufficient BRAM. Using this final precision and storage allocation strategy, all three TIPs can be synthesized onto the same FPGA in Vivado, enabling all 9 convolution layers to be supported. The final resource utilization is shown in Figure 4-12, confirming the proposed TIP-sharing method significantly increases the number of convolution layers that can be executed on the FPGA. Figure 4-13 shows the actual hardware interconnection in Vivado after implementing all three TIPs, as described in Section 3.1.

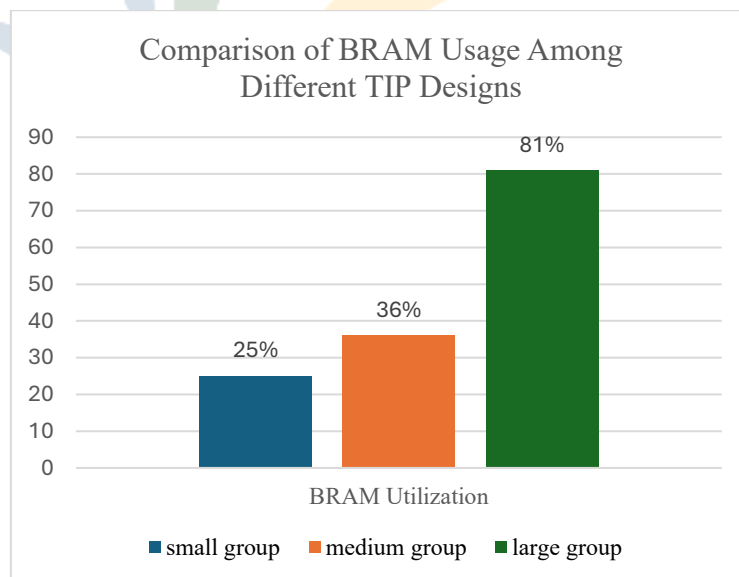


Figure4- 10 Comparison of BRAM Usage Among Different TIP Designs

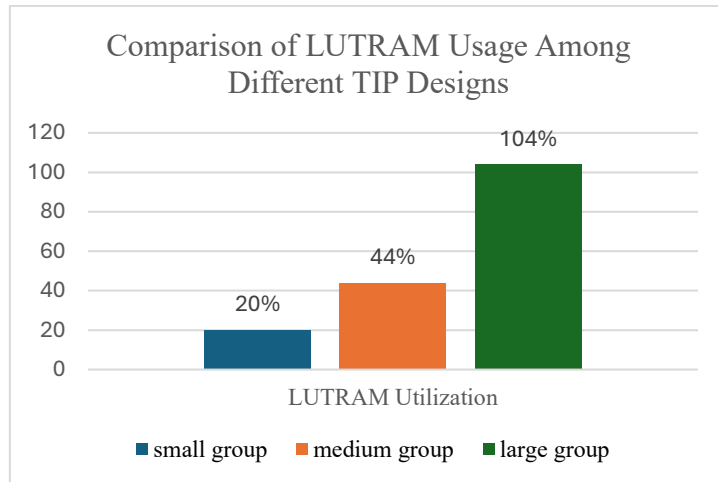


Figure4- 11 Comparison of LUTRAM Usage Among Different TIP Designs

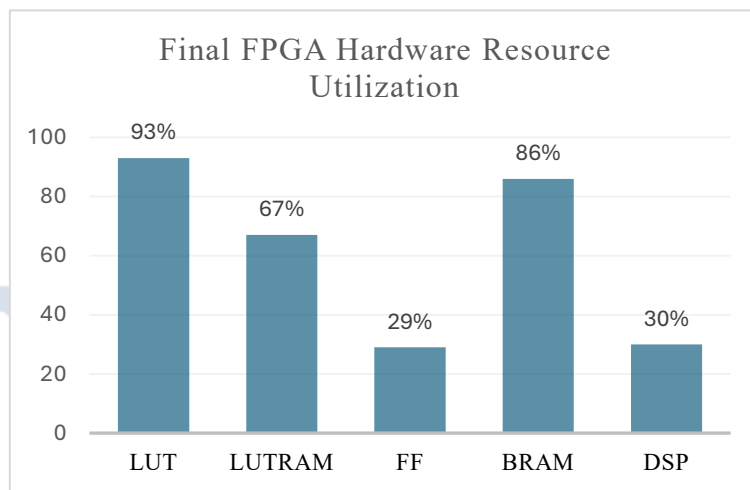


Figure4- 12 Final FPGA Hardware Resource Utilization

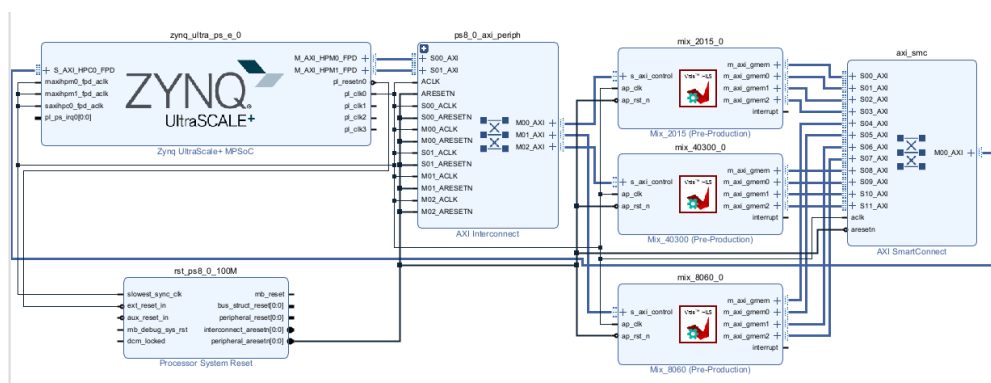


Figure4- 13 TIP Integration Results in Vivado

### 4.3 Summary of Execution Results

This study ultimately implemented all three TIPs on the FPGA simultaneously, enabling the computation of nine different convolution layers. In the YOLOv9-tiny model, 84 convolution layers can be accelerated using the FPGA, and their execution speed was compared with that of the same layers implemented using the traditional Conv2d function. By incorporating the `ap_fixed` format in the Vitis HLS design to adjust data precision, this thesis further analyzes and compares performance under different precision settings, evaluating actual object detection accuracy and mAP on the COCO128 dataset. Figure 4-14 shows the averaged results from 10 test runs before and after accelerating the 84 convolution layers on the FPGA. To further compare computational stability, standard deviation was also calculated as a supplementary evaluation metric. Compared with the default PyTorch Conv2d executed on four Cortex-A53 processors in the PYNQ-ZU platform, the FPGA-based implementation achieved a 54.37% reduction in execution time. Additionally, the standard deviation was significantly lower on the FPGA at 0.11 ms, compared to 2.48 ms under the ARM-based architecture. These results validate the FPGA's efficient parallel processing capabilities, which can not only shorten computation time but also enhance system-level performance stability. Figure 4-15 presents the mAP results across different precision settings, showing a clear trend of declining mAP as precision decreases. Based on comprehensive experimental analysis, `ap_fixed<18,6>` was selected as the final precision configuration for achieving a balance between hardware resource usage and model accuracy. Figures 4-16 to 4-19 show the object detection results under different precision settings. The inference using `ap_fixed<18,6>` achieved nearly identical detection performance to float32, with only minor differences—for instance, a detection confidence score dropped from 0.93 to 0.92 for one person in the bottom-left



corner. However, when the precision was reduced to  $\text{ap\_fixed}\langle 17,6 \rangle$ , a more noticeable performance drop occurred, with lower detection scores for most objects. At  $\text{ap\_fixed}\langle 16,6 \rangle$ , accuracy degraded significantly, indicating that excessive reduction in numerical precision adversely affects object detection performance.

Category	Without FPGA Acceleration	With FPGA Acceleration	Execution Time Reduction
Execution Time	530.37ms	241.99ms	54.37%
Standard Deviation	2.48ms	0.11ms	

Figure4- 14 Performance and standard Deviation in 84 Conv Layers

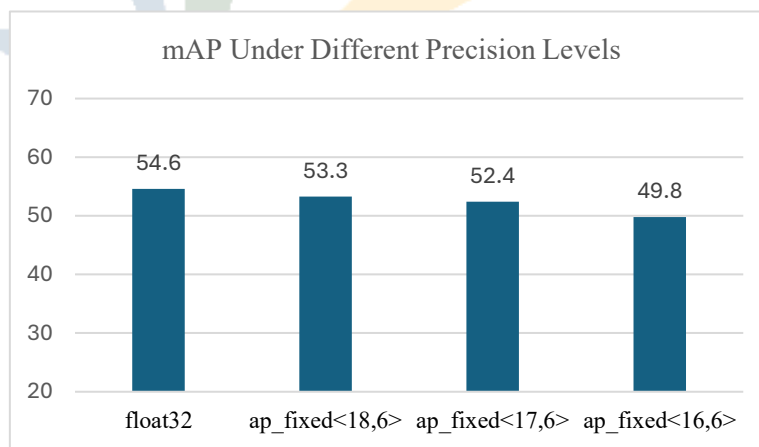


Figure4- 15 mAP Under Different Precision Levels (coco128)



Figure4- 16 Recognition Results with float32 Precision



Figure4- 17 Recognition Results with ap\_fixed<18,6> Precision



Figure4- 18 Recognition Results with  $ap\_fixed<17,6>$  Precision



Figure4- 19 Recognition Results with  $ap\_fixed<16,6>$  Precision

# Chapter 5: Conclusion and Future Work

## 5.1 Conclusion

The objective of this study is to leverage the highly parallel computing capabilities of FPGA to accelerate image recognition tasks in the YOLOv9 model, with a particular focus on optimizing the model's core convolution operations. The proposed architecture is designed to deliver both high performance and efficient resource allocation. During the design process, various directives provided by Vitis HLS, such as DATAFLOW and UNROLL, were utilized to achieve parallel execution and improve overall system throughput. Additionally, by adjusting the data storage precision, the usage of BRAM and LUTRAM was significantly reduced. Strategic allocation of BRAM and LUTRAM resources further enhanced FPGA storage efficiency and minimized hardware waste. To address the challenge of limited hardware resources, which prevent full offloading of all convolution layers to the FPGA, this study introduced a TIP-sharing strategy. This method allows multiple convolution layers to share the same TIP's BRAM or LUTRAM storage space, thereby reducing redundant design and memory consumption. The proposed strategy not only increases the number of convolution layers that can be accelerated but also enhances the scalability of the overall architecture. As a result of combining these technologies and optimizations, the FPGA-based convolution acceleration architecture developed in this work can be applied to all 84 convolution layers in YOLOv9-tiny, achieving a speedup of approximately 54.37% compared to the original implementation. The proposed approach could potentially be extended to other YOLO models that rely on Conv2d operations or even to other deep learning models

using Conv2d. However, actual implementation on other models would require further resource-specific optimizations based on the unique characteristics of each model's convolution layers.

## 5.2 Future Work

The EPIC architecture proposed in this thesis demonstrates significant acceleration when computing individual convolution layers compared to the Conv2d function in PyTorch. However, when applied to the entire YOLOv9 model, certain convolution layers cannot be accelerated due to hardware resource limitations on the FPGA, resulting in a diluted overall speedup. Most of the unaccelerated layers use kernels of size  $K_1=K_2=3$ , which consume significantly more resources for data storage and computation compared to layers with  $K_1=K_2=1$ . This excessive resource usage makes it more difficult to cover additional layers and achieve higher acceleration performance. Future research directions could focus on further optimizing the hardware resource consumption of  $K_1=K_2=3$  convolution layers, thereby allowing more layers to benefit from FPGA acceleration and improving overall performance. Alternatively, exploring new convolution algorithms that inherently require fewer hardware resources could help expand the number of layers that can be accelerated on FPGA platforms.

# References

- [1] C.-Y. Wang, I.-H. Yeh, and H.-Y. M. Liao, “YOLOv9: Learning what you want to learn using programmable gradient information,” arXiv:2402.13616, Feb. 2024.
- [2] C. Xu, S. Jiang, G. Luo, G. Sun, N. An, and G. Huang, “The case for FPGA-based edge computing,” *IEEE Trans. Mobile Comput.*, vol. 21, no. 7, pp. 2610–2619, 2022.
- [3] J. Jiang, Y. Zhou, Y. Gong, H. Yuan, and S. Liu, “FPGA-based acceleration for convolutional neural networks: A comprehensive review,” arXiv:2505.13461, May 2025.
- [4] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, “Learning transferable visual models from natural language supervision,” arXiv:2103.00020, Mar. 2021.
- [5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.
- [6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” arXiv:1506.02640, Jun. 2015.
- [8] J. Redmon and A. Farhadi, “YOLOv3: An incremental improvement,” arXiv:1804.02767, Apr. 2018.
- [9] R. Khanam and M. Hussain, “What is YOLOv5: A deep look into the internal features of the popular object detector,” arXiv:2407.20892, Jul. 2024.
- [10] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, “YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,” arXiv:2207.02696, Jul. 2022.



- [11] I. Logeswaran, H. Eissa, and R. Almadhoun, "Autonomous vehicle pedestrian detection & traffic sign recognition using YOLOv8," in *Proc. 29th Int. Conf. Automation and Computing (ICAC)*, 2024.
- [12] R. Bai, G. Xu, and Y. Shi, "SCC-YOLO: An improved object detector for assisting in brain tumor diagnosis," arXiv:2501.03836, Jan. 2025.
- [13] Q. Zheng, Z. Luo, M. Guo, X. Wang, R. Wu, Q. Meng, and G. Dong, "HGO-YOLO: Advancing anomaly behavior detection with hierarchical features and lightweight optimized detection," arXiv:2503.07371, Mar. 2025.
- [14] J. Moosmann, P. Bonazzi, Y. Li, S. Bian, P. Mayer, L. Benini, and M. Magno, "Ultra-efficient on-device object detection on AI-integrated smart glasses with TinyissimoYOLO," arXiv:2311.01057, Nov. 2023.
- [15] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," 2017.
- [16] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," arXiv:1511.08458, Nov. 2015.

NTPU

## 著作權聲明

論文題目：基於 PYNQ-ZU 之 YOLOv9 卷積層加速設計

論文頁數：63 頁

系所組別：通訊工程學系

研究生：劉哲安

指導教授：白宏達

畢業年月：114 年 6 月

本論文著作權為劉哲安所有，並受中華民國著作權法保護。

