MASTER THESIS

Real-Time YOLOv4 FPGA Design with Catapult High-Level Synthesis

Luuk Heinsius

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE
COMPUTER ARCHITECTURE FOR EMBEDDED SYSTEMS

EXAMINATION COMMITTEE
Dr. Ir. S.H. Gerez
Dr. Ir. N. Alachiotis
Dr. Ir. L.J. Spreeuwers

18-06-2021

UNIVERSITY OF TWENTE.

ABSTRACT

State-of-the-art object detectors play a vital role in identifying and localizing objects in images, especially during recent years with the up-rise of autonomous systems. This work develops a FPGA-based design for the real-time deep neural network (DNN) based object detector called YOLOv4. The design is targeting the ZedBoard which integrates a Xilinx Zynq-7020 SoC. A single-core bare-metal application integrating the TensorFlow Lite Micro (TFLM) framework provides a base platform to run a quantized version of YOLOv4. Convolutional layers, taking 99.67% of the total execution time, are speed up by a proof-of-concept accelerator. The accelerator has been designed based on the existing Eyeriss accelerator architecture [1][2]. The accelerator is implemented using High-Level Synthesis (HLS) C++ and gets synthesized to RTL via the Catapult HLS Platform. Integrating the accelerator with the TFLM framework shows speedups of convolutional layers of up to 11.67 times, a drop in energy consumption by a factor of 2.73, and bit-accurate accuracy compared to the original algorithm. Although a speedup is realized, real-time performance is not achieved. This is because of the complex architecture of the Eyeriss accelerator in combination with the limited time set for this project and the limited resources available on the FPGA.

CONTENTS

List of Abbreviations

AGEN Address GENerator.

AI Artificial Intelligence.

AMBA Advanced Microcontroller Bus Architecture.

ANN Artificial Neural Network.

AP Average Precision.

AXI Advanced Extensible Interface.

BN Batch Normalization.

CAD Computer-Aided Design.

CCOV Catapult Coverage.

CNN Convolutional Neural Network.

CONV Convolution.

CPU Central Processing Unit.

CSP Cross Stage Partial.

DNN Deep Neural Network.

DRAM Dynamic Random-Access Memory.

FC Fully Connected.

FPGA Field-Programmable Gate Array.

FPS Frames Per Second.

GIN Global Input Network.

GLB Global Buffer.

GON Global Output Network.

GPU Graphics Processing Unit.

HLS High-Level Synthesis.

Ifmap Input Feature Map.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.

IO Input/Output.

IoU Intersection over Union.

IP Intellectual Property.

LN Local Network.

lwIP LightWeight IP.

MAC Multiply And Accumulate.

mAP Mean Average Precision.

MC Multicast Controller.

MS COCO Microsoft Objects in COntext.

NN Neural Network.

Ofmap Output Feature Map.

OS Operating System.

PANet Path Aggregation Network.

PE Processing Element.

PL Programmable Logic.

PS Processing System.

Psum Partial Sum.

RF Register File.

RS Row Stationary.

RTL Register Transfer Level.

SAM Spatial Attention Module.

SIMD Single Instruction Multiple Data.

SLEC Catapult Sequential Logic Equivalence Checking.

SoC System-on-Chip.

SPad Scratch Pad.

SPP Spatial Pyramid Pooling.

TDP Thermal Design Power.

TF TensorFlow.

TF-Lite TensorFlow Lite.

TFLM TensorFlow Lite Micro.

YOLO You Only Look Once.

1 INTRODUCTION

Nowadays, computer vision is an active field of research showing impressive results. A popular computer vision task is object detection. Object detection enables systems to localize and classify objects in images. Traditional object detection methods relied on handcrafted feature extractors. These methods lag behind current methods using deep learning. One approach to applying deep learning that showed real-time performance for detecting objects was presented with the YOLO [3] (You Only Look Once) detector in 2016. YOLO presented a fresh approach where locations and corresponding classes were predicted straight from image pixels. Earlier techniques applied complex pipelines that are hard to optimize and perform relatively poorly. Multiple versions of YOLO have been published over the years, the latest scientific supported version is used in this work, which is version four (YOLOv4) [4].

Deep learning applications are commonly run on general-purpose processors such as CPUs and GPUs. Although providing a flexible computing platform which is beneficial for development, they no longer deliver sufficient processing throughput and energy efficiency [5]. As a result, developers optimize and accelerate their systems by designing dedicated hardware accelerators.

Designing hardware accelerators for such systems is complex in terms of design, implementation, and verification. Implementing these systems at the RTL level is therefore extremely challenging. The Catapult High-Level Synthesis (HLS) Platform from Mentor Graphics provides an easier approach by designing and verifying the system at C, C++, or SystemC level. Using this higher level of abstraction, compared to RTL, reduces the lines of code up to 80% [6] making HLS code easier to write and debug. Hardcoding specification in RTL such as parallelism and design throughput is avoided by allowing the designer to define these specifications using the Catapult interface. Another important fact is the HLS verification 100-500x speedup at the C level compared to RTL [6]. All this reduces complete industrial project time by half [7].

1.1 Problem Definition

The goal of this thesis is to develop a real-time YOLOv4 FPGA implementation with Catapult. Development is targeted on the ZedBoard[1], which is an ARM/FPGA SoC development board. To demonstrate the implementation, an application will be developed around YOLOv4 using a camera and a screen. Captured images are fed into the YOLOv4 object detector and, after processing, post-processed to be overlaid on the original image and streamed to a screen. The points below summarize the tasks to be performed on the system:

1. Image capture
2. Video streaming
3. YOLOv4 algorithm processing

[1] `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/zedboard-board-family`

4. Pre-processing (image re-scaling, etc..)
5. Post-processing (prediction filtering, drawing bounding boxes, etc..)

Two system designs were considered, one of which realizes all tasks on the ZedBoard, and the other uses a combination of a host PC and the ZedBoard. The ZedBoard will then only do the YOLOv4 algorithm processing and all other steps should be taken care of by the host PC. The last design introduces the additional task of interfacing both systems but focuses more on the YOLOv4 FGPA design. For this last reason, the second design was chosen. This removes the implementation of the image capture and the video streaming IP blocks. This saves time, which is already limited by the six months set for the project. The removal of the two IPs also relaxes the area constraints. An overview of the system is presented in Figure 1.1.

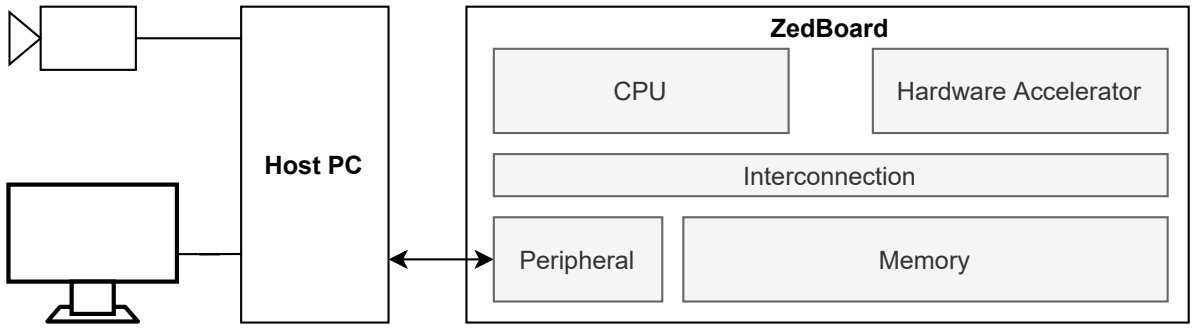


Figure 1.1: System overview where the YOLOv4 algorithm processing is performed on the ZedBoard and all other processing is taken care of by the host PC.

1.2 Approach

Since a limited time frame is set for this project, it is essential to narrow down the design space on how to design/implement the YOLOv4 algorithm on the ZedBoard as quickly as possible. It has therefore been decided that the YOLOv4 model will run on the CPU using a deep learning framework with bottle-neck functions being hardware accelerated. This has the additional advantage that other models supported by the framework can be accelerated on this system.

1.3 Research Questions

The main research question is formulated as follows:

Can a real-time FPGA design be created with the Catapult High-Level Synthesis Platform for the deep learning object detector YOLOv4 on the ZedBoard?

To answer the main research question, it is divided into multiple research sub-questions:

1. *Which deep learning framework can be best used for creating the software application?*
2. *Which part(s) of the software application can be hardware accelerated?*
3. *Can the YOLOv4 model be optimized before designing a hardware accelerator?*
4. *How can a YOLOv4 accelerator be created using the Catapult High-Level Synthesis Platform?*
5. *How can the interface between the host PC and the System be implemented?*

1.4 Contributions

The goal, as formulated in the main research question, is to create a real-time FPGA design with the Catapult High-Level Synthesis Platform for YOLOv4 targeting the ZedBoard. However, this is not the only contribution of this work. The main contributions of this thesis have been listed below:

- Single-core bare-metal software application integrating the TensorFlow Lite Micro (TFLM) framework providing a base platform to run neural networks on the ZedBoard (Section 5.1).
- Workflow to quantize a TensorFlow model, convert it to a compatible TFLM model, and cross-compile the software application together with a model that allows it to be run on the ZedBoard (Section 5.1.2).
- Highly configurable FPGA-based hardware accelerator for convolutional layers of the TFLM framework implemented in High-Level Synthesis C++ (Chapter 6).
- Configurator allowing users to configure the accelerator (Section 6.5).
- Synthesis of the accelerator using the Catapult High-Level Synthesis Platform (Section 6.7).

1.5 Outline

The further chapters of the report are organized as follows:

- **Chapter 2** provides an introduction to deep neural networks (DNNs), object detection, and DNN frameworks.
- **Chapter 3** describes YOLOv4 in detail, how to post-process the predictions, and related work that use YOLO.
- **Chapter 4** introduces the most important features of the Catapult High-Level Synthesis tool.
- **Chapter 5** analyzes which part of the software application can be best accelerated in hardware. This is done by first describing how the software application is implemented and then after profiling, analyzes the function taking the most execution time.
- **Chapter 6** contains a comprehensive explanation of how the previously identified bottleneck function is accelerated by first designing a hardware accelerator and then implementing it. It also describes how the accelerator is synthesized using Catapult and how the generated files are used for final system integration.
- **Chapter 7** describes the results of the system and evaluates them.
- **Chapter 8** finally presents the conclusions that have been drawn from this work.

Please note that the first two chapters, i.e., Chapter 2 and Chapter 3, serve as background information. These chapters might be skipped if the reader is already familiar with DNNs and YOLOv4.

2 DEEP NEURAL NETWORKS

Deep Neural Networks (DNNs) are a small subset of the artificial intelligence (AI) field and are often referred to as deep learning (DL). AI attempts to understand and build intelligent entities and was coined in the 1950s [8]. In Figure 2.1 the relationship of DNNs in the field of AI is visualised.

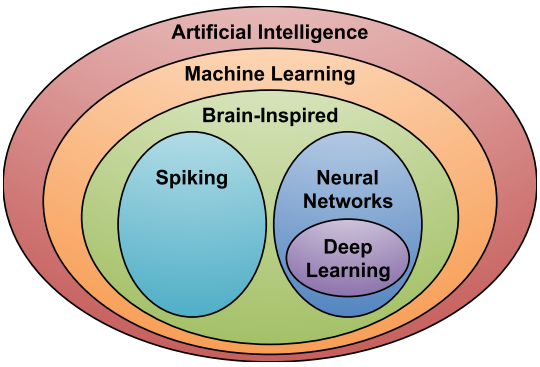


Figure 2.1: Deep Learning in the AI context [9].

This chapter first introduces, in Section 2.1, the general aspects of artificial neural networks. Then, the DNN application type used in this work called object detection is introduced in Section 2.2. Finally, in Section 2.3, existing frameworks for the development of DNNs are elaborated.

2.1 Introduction

Artificial neural networks (ANNs), typically called neural networks (NNs), are inspired by the findings of neuroscience and in particular, the hypothesis that mental activity consists primarily of electrochemical activity in a network of brain cells called neurons. Figure 2.2 displays the mathematical representation of a neuron.

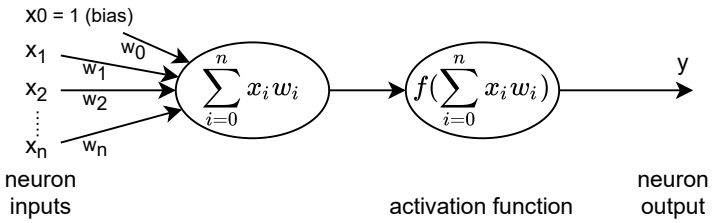


Figure 2.2: Mathematical model of a neuron.

Each neuron has a vector of n inputs $\mathbf{x} = [x_0, x_1, ..x_n]$. The first input x_0 is called the bias, and its value is constant, leaving only $n-1$ controllable inputs. Each input connects to a neuron via a link. Each link has a numeric weight w_i associated with it. So in combination with n inputs, we have a vector of n weights $\mathbf{w} = [w_0, w_1, ..w_n]$. A neuron computes its output by applying a

differentiable activation function to the weighted sum of the inputs, see equation 2.1. Section 2.1.1 provides an in-depth look into the existing activation functions.

$$y = f(\sum_{i=0}^{n} x_i w_i) \tag{2.1}$$

Neural networks are created by connecting multiple neurons. Two types of networks exist: feed-forward networks and recurrent networks. Feed-forward networks connect all neurons in one direction and form a directed acyclic graph. Information in this network moves in one direction from the input to the output, and the network has no internal state. Recurrent networks, on the other keep their state by connecting the outputs back to the inputs.

Figure 2.3 depicts the structure of a feed-forward neural network. The network is arranged in layers where each layer receives the input from the previous layers. Nodes in the input layer represent the input data. The output is obtained by propagating the input data through the network until it reached the output layer. All layers between the input- and output layers are called hidden layers. Note that each layer connects a bias node to the next layer.

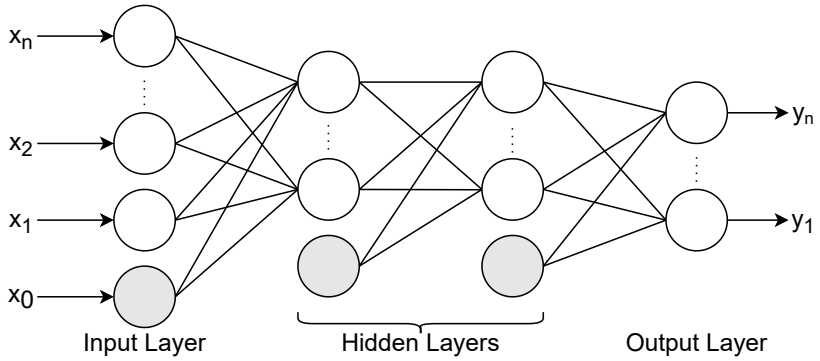


Figure 2.3: Feed-forward neural network example with three input nodes, two hidden layers with each two neurons, an output layer made of two nodes. The grey nodes represent the bias nodes.

2.1.1 Activation Functions

Activation functions compute the output of a neuron with the weighted sum of the inputs. This section presents some of the well-known activation functions. Figure 2.4 graphically shows these functions.

- **Sigmoid**
  The traditional sigmoid function, see equation 2.2, has been used for many years [10] and is one of the most common forms of activation functions [11].

  $$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

  Deep neural networks do not use this function often except for the output layer owing to its value distribution.

- **Hyperbolic Tangent**
  Hyperbolic Tangent, defined in equation 2.3a, can be easily deducted from the sigmoid function, see equation 2.3b.

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3a}$$

$$tanh(x) = 2sigmoid(x) - 1 \tag{2.3b}$$

  The Hyperbolic Tangent is more preferred than the sigmoid function because of its symmetry around the origin, which leads to the output being on average close to zero. Also, the classification error of networks that use the Hyperbolic Tangent is lower than those that use the sigmoid activation function [11]. One disadvantage compared to the sigmoid function is its relatively complex derivative needed for training.

- **Rectified Linear Unit (ReLu)**
  The Rectified Linear Unit (ReLu) function, equation 2.4, is currently almost the most popular activation function used in deep neural networks [11]. Some of the advantages [11] are: 1) computation is cheaper than sigmoid and hyperbolic tangent, 2) neural networks converge faster compared to saturating functions, 3) the derivative of ReLu is one which avoids local optimization and resolves the vanishing gradient effect[1], and 4) a sparse[2] representation is easily obtained.

$$f(x) = \begin{cases} 0 & for \quad x \leq 0 \\ x & for \quad x > 0 \end{cases} \tag{2.4}$$

  Deactivated neurons because of sparsity form a disadvantage since this leads to the death of neurons. These dead neurons always produce the same output because all inputs get multiplied by zero and therefore take no role in producing usable results. Another disadvantage is that a bias shift can be introduced because of the output being identically positive.

- **Leaky ReLu**
  Leaky ReLu, defined in equation 2.5, is an adapted version of the ReLu activation function. The goal of Leaky ReLu is to prevent dead neurons by multiplying x with a small positive scalar.

$$f(x) = \begin{cases} ax & for \quad x \leq 0 \\ x & for \quad x > 0 \end{cases} \tag{2.5}$$

- **Mish**
  Mish [12] was proposed to improve performance and address the shortcomings of ReLU, just like Leaky ReLu. The researchers of Mish found that Mish matches or even improves the performance of neural networks as compared to that of ReLu and Leaky ReLu across different tasks in computer vision. Equation 2.6a defines the Mish activation function mathematically.

$$f(x) = x \cdot tanh(softplus(x)) \tag{2.6a}$$

$$softplus(x) = ln(1 + e^x) \tag{2.6b}$$

[1] More information on the vanishing gradient effect can be found in Section 2.1.3.
[2] Sparsity implies that the vast majority of the weights are 0.

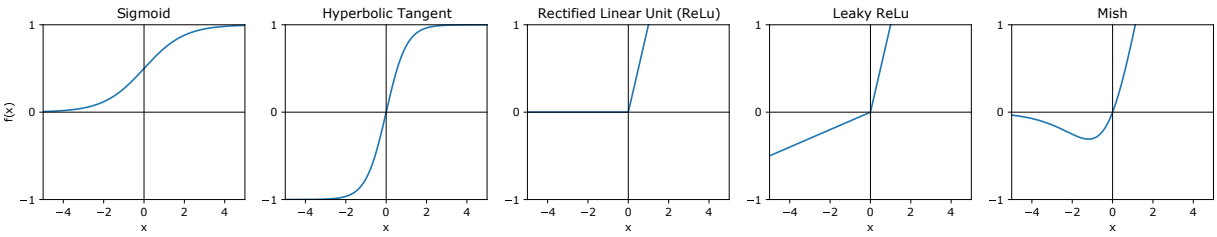


Figure 2.4: Nonlinear activation functions commonly seen in neural networks.

2.1.2 Network Training

Neural networks belong to the machine learning field, implying that the network needs to able to learn. Learning involves adjusting the weights of the network to minimize the computed and expected network output. The most used approach for learning the network is called supervised learning. Supervised learning tries to optimize the weights by feeding the network with labeled training data. Now that the output is known, the prediction error $E(\mathbf{w})$ can be computed.

Most techniques initialize the weight vector $\mathbf{w}^{(0)}$ and then move through the weight space in a succession of steps τ in the form:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \triangle \mathbf{w}^{(\tau)}$$

Many algorithms exist for updating the weight vector with weight vector update $\mathbf{w}^{(\tau)}$. The most popular algorithm is Stochastic Gradient Descent [13] and updates the weight vector with the gradient of the error function:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \bigtriangledown E(\mathbf{w}^{(\tau)})$$

Parameter $\eta > 0$ is called the learning rate and must be carefully selected to prevent slow converging or even failure to converge due to η being too large. The error function calculates for each step the error over the entire training data set (training epoch).

2.1.3 Backpropagation

The backpropagation process adjusts all weights in a feed-forward neural network. Backpropagation is an iterative procedure that tries to minimize the error function $E(\mathbf{w})$ by first computing the error (forward pass), and then adjust the weights in a sequence of steps. Each step requires two stages: 1) calculate the gradient of the error function with respect to the weights (backward pass), 2) use the gradient error to adjust the weights (update phase). This process continues until all errors as calculated in stage one are propagated backward through the network.

A problem that can be encountered during this process is called the vanishing gradient problem. This problem may arise in neural networks with a lot of layers resulting in the gradient of the error becoming smaller and smaller during the backward pass. Eventually, the gradient will be extremely small, preventing the weights from being updated.

2.1.4 Layer Types

DNNs comprise multiple layers that each can have different functionality. Layer types can be categorized into two groups: layers whose main computation is a weighted sum and layers that do not use a weighted sum. This section summarizes some popular layer types. The first two layer types use a weighted sum, the other types following do not use it.

Fully Connected Layer

Fully connected layers connect all neurons from one layer to all neurons in another layer. The main computation is a weighted sum of the inputs. Convolutional neural networks typically use one or more fully connected layers for decision making.

Convolutional Layer

Convolutional layers process 2D data such as images. A key property of images is that nearby pixels are more strongly correlated than more distant pixels. Therefore, convolutional layers try to extract local features that rely only on small subregions of the image. This small subregion commonly known as the receptive field defines the region in the input space that a particular layer is looking at. Because of this property, using a fully connected layer to process images results in key properties of the image being ignored.

Data is organized into planes which are called feature maps. The layer receives 3D input feature maps consisting of ch_{in} channels and 2D images of dimension $h_{in} \cdot w_{in}$. The channels represent different channels used in images such as the RGB channels or the intensity of a pixel. Processing the input feature maps gives the output feature maps with ch_{out} channels and 2D images of dimension $h_{out} \cdot w_{out}$. The output feature maps are created by the convolution of the input feature maps and convolutional kernels, which represent the weights of the layer. These kernels are small filters of size $k \cdot k$ and have the same amount of channels as the input feature maps. Each input feature map undergoes a 2D convolution with its corresponding kernel channel. All convolution results for each channel are then accumulated to generate the output feature map. Multiple output feature maps can be created by using additional 3D kernels ch_{out}. Figure 2.5 summarizes the theory presented above.

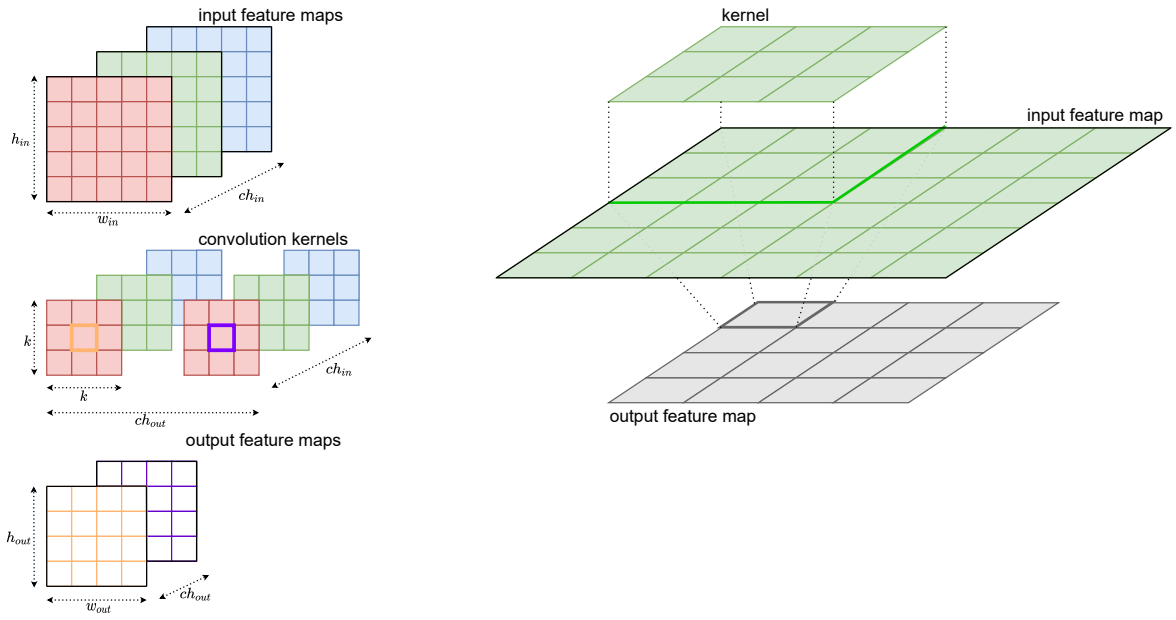


Figure 2.5: Left: ch_{in} input feature maps (RGB) are convolved with $ch_{in} \cdot ch_{out}$ kernels with size $k \cdot k$. This result in ch_{out} output feature maps (G,P). Right: Output feature map computation example by sliding the kernel over the input feature map. Figure adapted from [14].

The amount by which the kernel slides over the input feature map is defined by a term called *stride*. Setting *stride* to n means that each shift (x or y) moves n place(s).

Pooling and Unpooling Layer

Convolutional neural networks commonly use pooling layers after a convolutional layer. Pooling reduces the dimension of the data by removing irrelevant details. This also makes the convolution features robust to minor variations in the input [15]. Figure 2.6 demonstrates two pooling strategies commonly found in the literature. Max pooling compresses a block with n by m dimensions by taking the maximum value. Average pooling also takes a block but averages all values.

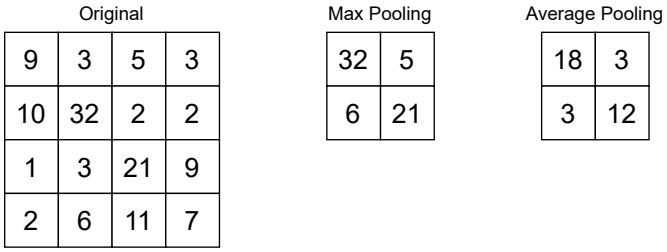
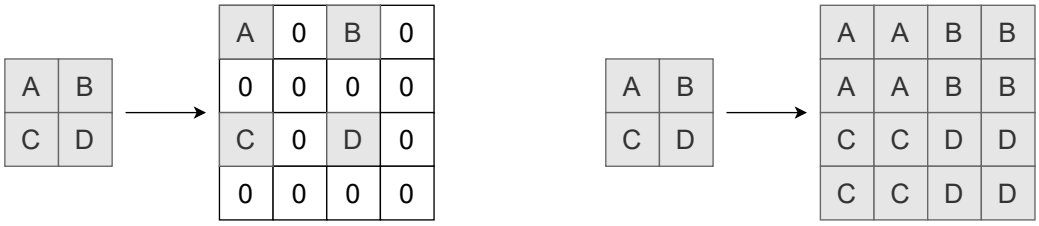


Figure 2.6: Max- and average 2x2 pooling example with stride=2.

Unpooling layers increase the dimension (upsampling) of the data. These are usually placed before convolutional and fully-connection layers to introduce structured sparsity [9]. Two common unpooling techniques are depicted in Figure 2.7.



(a) Zero-insertion.

(b) Nearest neighbor.

Figure 2.7: Two unpooling techniques.

Normalization Layer

Reducing the training time of neural networks and improving accuracy can be achieved by normalizing the layer output distribution [16]. This is especially useful for shifts introduced by, for example, the ReLu activation function. A normalization layer can reduce this shift by fixing the mean and the variance of all summed inputs of that layer. Consider the vector of summed inputs a^l of layer l and H denoting the number of hidden neurons in l then the layer normalization statistics are as follows:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \tag{2.7a}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{2.7b}$$

Nonlinearity Layers

Layers that use the weighted sum for its main computation typically use a nonlinearity layer at the output. See Section 2.1.1 for more in-depth information.

Dropout Layers

The dropout layer was introduced to prevent overfitting in neural networks [17]. During the training phase, neurons and all their connections are removed (dropped) from the network. Dropping out neurons is performed randomly. As an impact of dropping neurons, abstraction is forced, preventing the network to learn very precise mappings.

2.2 Object Detection

Object detection is a popular application type of DNNs. Detecting objects consists of two tasks: one is the object localization and the second is the classification of objects. Object localization indicates the location of objects by spatially separated bounding boxes around them. Object classification predicts the class of the detected object. State-of-the-art detectors utilize deep learning networks as their backbone for feature extraction on input images and a detection network for localization and classification. These networks are classified as convolutional neural networks (CNNs) and elaborated in Section 2.2.1. Section 2.2.2 covers the evaluation metrics used for evaluating the accuracy of object detectors. Finally, the datasets used for object detection, specifically for YOLOv4, are described in Section 2.2.3.

2.2.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are widely applied to image data and are commonly used for tasks like object detection, object tracking, scene labeling, speech recognition, and many more [9]. These networks mainly comprise convolutional layers to extract local features from the image. It then merges extracted features in later stages of processing to obtain a higher abstraction and finally yield information about the image. The common structure of CNNs is depicted in Figure 2.8.

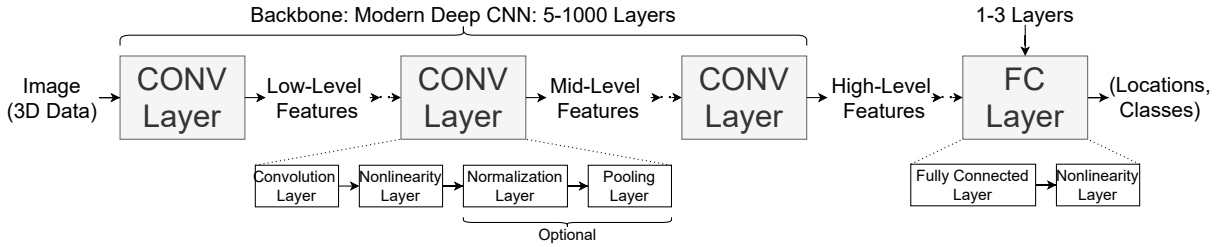


Figure 2.8: Convolutional neural network basic structure. Figure adapted from [18].

After each convolutional layer, a nonlinearity layer transforms the data. Optionally the data is then processed by a normalization layer and/or a pooling layer to subsample the data. The final layer of the network would typically be fully connected with a nonlinearity layer in the case of localization and classification.

2.2.2 Evaluation Metrics

The accuracy of object detectors is determined by the quality of localization and classification of objects. Measuring the accuracy of object detectors is commonly performed using two popular metrics: Average Precision (AP) and Mean Average Precision (mAP). Datasets for object detection usually adapt these metrics, therefore this section describes only the basis of these metrics. For the exact metrics used in YOLOv4 see Section 2.2.3.

This section first describes the fundamental concepts of precision, recall, and Intersection over Union (IoU). Next, classifying prediction using these metrics is elaborated. Finally, the two popular metrics are explained.

Precision and Recall

Precision measures how accurate the prediction is, i.e., the ratio of true positive tp and the total number of predicted positives. Equation 2.8 mathematically defines precision, where the false positives are indicated by fp.

$$Precision = \frac{tp}{tp + fp} \tag{2.8}$$

The disadvantage of precision is that it does not consider predictions classified as negative that are positive in reality (false negative fn). Recall solves this by providing a metric between the ratio of tp and total of ground truth positives (Equation 2.9).

$$Recall = \frac{tp}{tp + fn} \tag{2.9}$$

Intersection over Union

The IoU metric measures how accurately a bounding box is predicted compared to the ground truth bounding box. Figure 2.9 illustrates how the IoU is calculated.

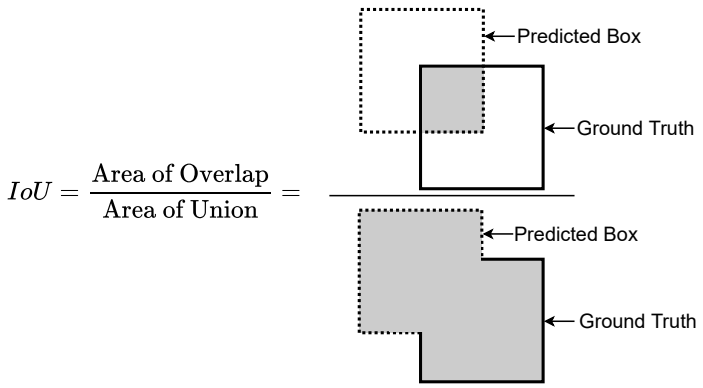


Figure 2.9: Intersection over Union (IoU). Area in gray.

Classifying predictions

When classifying predictions, we take both the classification and location into aspect. Classification determines if the right object class is predicted. For classifying the predicted location, we use the IoU and an IoU threshold. One aspect not yet presented but used in the classification of predictions is the confidence score. The confidence score defines the probability that an anchor box contains an object. See Section 3.2.1 for more information on anchor boxes.

The rules for classifying predictions are:

- True positive tp (all must apply):
  1. The confidence score is higher than the confidence threshold.
  2. The predicted class matches the class of a ground truth.
  3. The predicted bounding box has an IoU greater than the IoU threshold.
- False positive fp: Violation of either of the two latter conditions.
- False negative fn: The confidence score of detection is lower than the confidence threshold, but is supposed to detect a ground truth.
- True negative tn: The confidence score of detection that is not supposed to detect anything is lower than the confidence threshold.

Note that dataset challenges sometimes include additional rules as explained in Section 2.2.3.

Average Precision

The Average Precision (AP) metric encapsulates both precision and recall as a measure to evaluate the performance of object detectors for detecting a certain class. AP is defined by finding the area under the precision-recall curve across recall values from 0 to 1. The precision-recall curve is created by setting the confidence score at different levels and thereby generating different pairs of precision and recall. Figure 2.10 displays a precision-recall curve.

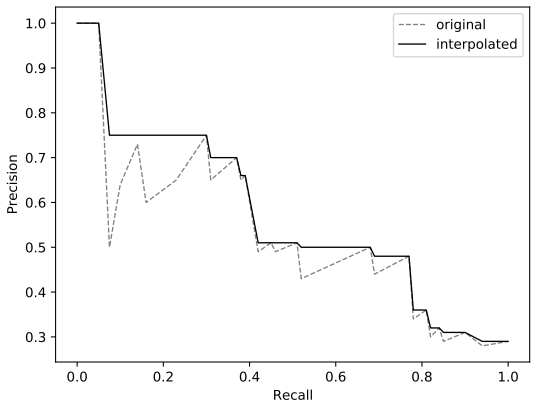


Figure 2.10: Precision-recall curve example. Gray dashed line: original curve. Black line: interpolated curve.

The AP is calculated by integrating the precision $p()$ with respect to recall r on interval $[0, 1]$, see Equation 2.10.

$$AP = \int_0^1 p(r)dr \tag{2.10}$$

Before calculating the AP, the precision is interpolated by taking the maximum precision value to the right at each recall level $r' \geq r$, see Figure 2.10. The interpolated precision $p_{interp}()$ at a recall level r is defined as:

$$p_{interp}(r) = \max_{r' \geq r} p(r') \tag{2.11}$$

Mean Average Precision

The AP metric calculates the average precision of the object detector on predicting one class. Mean Average Precision (mAP) on the other hand, averages AP over K classes. mAP is defined as:

$$mAP = \frac{1}{K} \sum_{i=1}^{K} AP_i \tag{2.12}$$

2.2.3 Datasets

YOLOv4 uses two datasets for training. First, the feature extractor of the model is trained separately on the ImageNet dataset and then the complete model on the Microsoft COCO dataset. This section covers both of these datasets.

ImageNet

A popular testbench for CNNs is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [19]. This annual challenge has been run from 2010 to the present and is a benchmark in object category classification and detection. ILSVRC consists of two components: a publically available dataset and an annual competition. The ImageNet dataset consists of over 14 million images, each labeled with one class. Contestants train their networks with a publically released dataset containing 1.2 million labeled images in 1000 distinct classes. A set of test images without annotations test the networks. Contestants submit their predictions to an evaluation server, and it reveals the results at the end of the competition. It measures accuracy in two forms: top-1 accuracy tracks the correct classified images at the first place (top 1), and top-5 accuracy is the percentage of classified images that were in the top 5 predicted classes.

Images are annotated using two categories: image-level annotations of a binary label defining the presence or absence of an object, and object-level-annotation of a tight bounding box and class label around an object instance.

Microsoft COCO

The Microsoft Common Objects in COntext (MS COCO) [20] dataset contains 80 object categories and has 330.000 images from which over 200.000 are labeled. In total, the dataset labeled 1.5 million object instances with multiple instances in a single image. Each instance is 2D localized enabling networks using this dataset to learn both classification and localization of objects. In contrast with ImageNet, COCO has fewer categories but more instances per category.

COCO evaluates accuracy using a modified mAP metric. The authors make no distinction between AP and mAP and simply call their metric AP which is traditionally called mAP. Recall is divided into 101 points for generating recall precision pairs. This results in the following Equation with n=101:

$$AP = \frac{1}{n} \sum_{r \in \{0, \frac{1}{n}, .., 1\}} p_{interp}(r) \tag{2.13}$$

Computing the AP is divided into three sub-metrics. The first sub-metric evaluates a model over ten IoU thresholds and averages the result. The last two use a fixed IoU threshold. Summarizing these sub-metrics:

1. AP :AP at $IoU = .50 : .05 : 0.95$
2. $AP^{IoU=.50}$:AP at $IoU = .50$
3. $AP^{IoU=.75}$:AP at $IoU = .75$

2.3 Frameworks

DNN frameworks provide implementations of common deep learning algorithms. Some frameworks also have pre-trained deep neural network models available. These tools allow the acceleration of development and research in the field. Frameworks work with a higher abstraction level that lets users define the skeleton of the application. Configuration files define the application skeleton that describes the layer types, neurons per layer, shape of input data, etc. Many frameworks offer the possibility to accelerate the inference and learning process by a GPU. YOLOv4 was originally implemented in the Darknet framework, but implementations in other frameworks exist. Finding a framework that helps to solve the problem the best is important, that's why, next to Darknet, two other popular frameworks are discussed in this section. Table 2.1 summarizes these frameworks.

Table 2.1: Popular deep neural network frameworks.

Framework	**Core Language**	**Binding(s)**	**Pre-trained Models**	**Developer(s)**
Darknet[21]	C and CUDA	Python	All YOLO versions and other models	Joseph Redmon
TensorFlow[22]	C++	Python, JavaScript Java, Go, Swift	MNIST, ResNet, EfficientNet, Retina, more in Model Garden	Google
Caffe[23]	C++	Python, MATLAB	CaffeNet, AlexNet, R-CNN, GoogLeNet	Berkeley AI Research

Darknet

Darknet [21], developed by the original YOLO author Joseph Redmon, is a deep learning framework supporting CPU and GPU computation. The documentation mainly consists of .readme files on GitHub and focuses only on basic information. This makes it difficult to be used in production environments. Models are defined in *cfg* configuration files and dynamically created at runtime. Network weights are stored in *weight* files. In addition to inference and training of models, Darknet can also perform AP and FPS evaluation.

Caffe

Convolutional Architecture for Fast Feature Embedding (Caffe) is developed by Berkeley AI Research (BAIR) and offers a modifiable framework for state-of-the-art deep learning algorithms [23]. Development and research are further sped up by popular pre-trained models such as AlexNet being available. The framework is written in C++ with Python and MATLAB bindings. Since the core language is written in C++, direct mapping the framework to different hardware platforms is possible. Models are defined in *prototxt* format. Weights are stored in a *caffemodel* format and the image mean of the data in *binary proto* format. The compiled framework uses these files to dynamically create the model at runtime.

TensorFlow

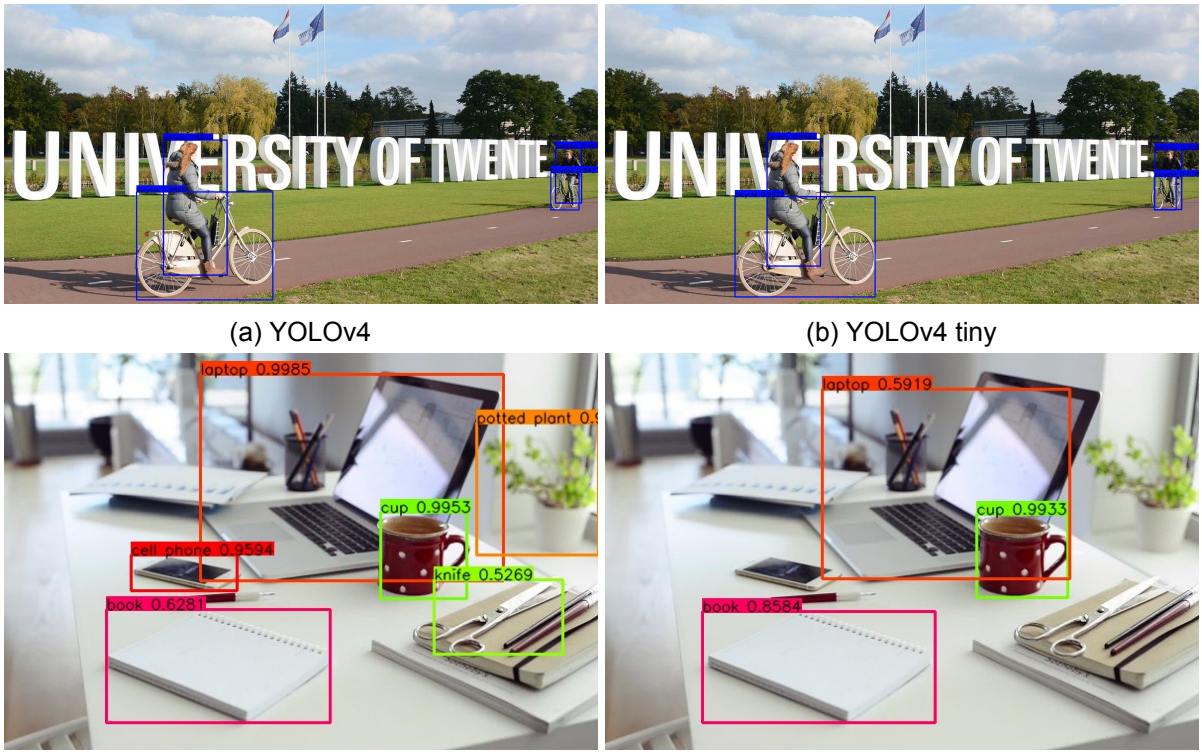
TensorFlow [22], short for Large-Scale Machine Learning on Heterogeneous Distributed Systems, is developed at Google by the Google Brain deep learning research team. Compared to the two other frameworks, TensorFlow is the most popular, has the most documentation, and an active community. High-level APIs such as Keras allow for easier development of models. Models, unlike Caffe and Darknet, are not defined in a configuration file but are described as a dataflow graph in code. TensorFlow allows the mapping of these models on different hardware platforms from CPU, one GPU to many GPU cards, to specialized machines with thousands of GPUs. Besides general-purpose computing devices, running and training models on their hardware accelerator (TPU) are supported.

Next to the hardware platforms described earlier, hardware platforms at the edge of the network such as mobile, embedded systems, and IoT devices are supported through a separate framework called TensorFlow Lite Micro (TFLM). Models in TFLM do not require operating support, any standard C or C++ libraries, or dynamic memory allocation. TFLM for microcontrollers is written in C++ 11 and requires a 32-bit platform.

Deploying models on a microcontroller can be realized by first creating the model in the easy to program Python TensorFlow environment and then convert it to TFLM. Another helpful feature of TFLM is the possibility to optimize a model. Optimization such as quantization, pruning, and clustering can be applied to improve both model size and inference speed.

3 YOLOV4

You Only Look Once version 4 (YOLOv4) [4] is a real-time CNN for object detection. The network predicts bounding boxes and class probabilities from images in one evaluation. The real-time aspects come from the fact that the detection is framed as a regression problem. As a result, there is no need for a complex pipeline system, so by simply running the network on an image, detections are predicted. There exist in total five versions of YOLO but only the first four [3][24][25][4] are supported by a scientific paper at the time of writing. Therefore, the latest scientific supported version is used, which is YOLOv4. YOLOv4 has been published on 23 April 2020. YOLOv4 comes with a tiny version that focuses on systems with limited resources. This tiny model applies the same techniques as used in YOLOv4 but has fewer convolutional layers. Figure 3.1 provides predictions of two different images comparing the accuracy of YOLOv4 and YOLOv4 tiny.



(a) YOLOv4 (b) YOLOv4 tiny

(c) YOLOv4 (d) YOLOv4 tiny

Figure 3.1: Difference between object detectors YOLOv4 and YOLOv4 tiny.

This chapter starts by summarizing all preceding versions of YOLOv4 in Section 3.1. Next, Section 3.2 describes the input and output of the network. This should give the reader a good understanding of the object detector. Section 3.3 provides a detailed description of the architecture. Post-processing of the predictions is elaborated in Section 3.4. Finally, Section 3.5 provides a short overview of related work using YOLO.

3.1 History

YOLOv1 [3] was first presented in May 2016 by the main researchers Joseph Redmon and Ali Farhadi and introduced an alternative approach to object detection. Prior work on object detection commonly used complex system pipelines in which first interesting locations in the input image were determined, then a classifier was used to classify objects in these locations. This complex pipeline is hard to optimize and performs poorly. YOLOv1 reframes object detection as a single regression problem, this means that localization and classification are performed straight from image pixels. This simplicity makes YOLO fast, computing 45 frames with no batch processing on a Titan X GPU. It also achieved more than twice the mAP compared to other real-time object detectors at the time.

YOLOv2 [24] was released in December 2016 and presented a better, faster, and stronger YOLO model. Batch normalization layers were added on all convolutional layers, which improved the mAP by more than 2%. Next, the classification network was trained on 448 x 448 resolution images compared to 224 x 224 in YOLOv1 increasing mAP by almost 4%. The original version predicted bounding box coordinates directly, by replacing this with bounding box priors and predicting offsets, the mAP dropped by 0.3% but an increase in recall from 81% to 88% proved that the model has more room to improve.

The classification network used in YOLOv1 was based on the Googlenet architecture using 8.52 billion operations for a forward pass. YOLOv2 makes use of a new model called Darknet-19. Darknet-19 has 19 convolutional layers and 5 max-pooling layers and required fewer operations (5.58 billion), making YOLOv2 faster than YOLOv1. The model was strengthened by using new training methods.

YOLOv3 [25], released in May 2018, extended the Darknet-19 classification network, renamed it to feature extractor, with residual connections, and added more layers. They named it Darknet-53 since it uses 53 convolutional layers. This network is much more powerful than Darknet-19 but increases operations by more than a factor of two.

YOLOv4 [4], released in April 2020, changed developers because the previous developers stopped their efforts in computer vision research. They were concerned about how the technology was being used for military applications and that the privacy concerns were having a societal impact. This version mostly combines state-of-the-art methods to improve YOLOv3.

3.2 Input and output

YOLOv4 processes input images with a resolution of N x N pixels and three channels. The pixel resolution N must be a multiple of 32. The authors of YOLOv4 used three different resolutions for their experiments, which are: $N = 416$, $N = 512$, and $N = 608$. A higher resolution input picture leads to a higher accuracy but also higher training and inference time. Most of the publicly available pre-trained YOLOv4 models are trained using the $N = 512$ resolution. The examples shown in this chapter use the $N = 416$ resolution.

The network predicts objects at three different scales. This means that feature maps are extracted at three different levels in the feature extraction point of the network. Since the feature extraction part consists mainly of convolutions, input images will get smaller and smaller by going deeper into the network. Thus by extracting feature maps at different points, high, medium, and small features are preserved. This is useful for detecting objects of different sizes, for example, cars are relatively large, so detection using small features (lower resolution) is favorable. On the other hand, detecting small objects such as traffic lights can be done by the high feature maps (high resolution). Figure 3.2 illustrates the idea of extracting features on different levels.

The size of an output stage N_i is defined at each stage i as:

$$N_1 = N_{in}/8\ ,\ N_2 = N_{in}/16\ ,\ N_3 = N_{in}/32 \tag{3.1}$$

Each output pixel in the output feature map, now referred to as a grid cell, is a 1D tensor[1] predicting an object's location and class. The 1D tensor consists of four predicted coordinates for each bounding box t_x, t_y, t_w, t_h and an objectness score p (confidence score). For more information on bounding boxes, refer to Section 3.2.1. Each 1D tensor also predicts C conditional class probabilities. This results in the tensor containing the following predicted tuple: $[(t_x, t_y, t_w, t_h), p_c, (C_1, C_2, .., C_n)]$. Since the output of stage i is made of N_i grids, we have a 3D tensor with N_i x N_i 1D tensors. These 3D tensors are known as boxes. Each stage predicts three boxes, see Figure 3.2.

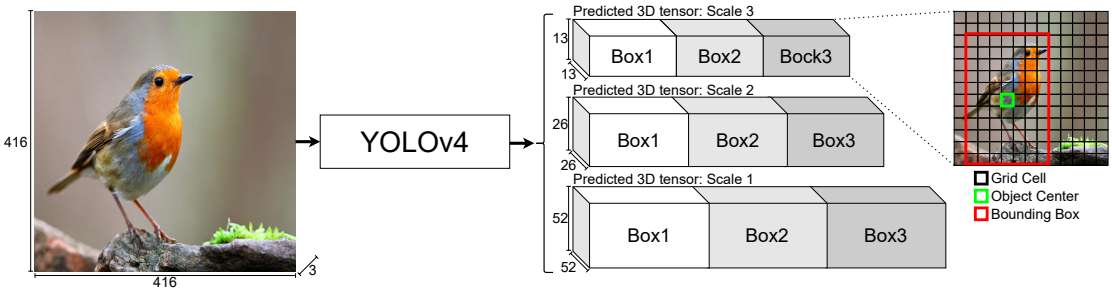


Figure 3.2: YOLOv4 process overview.

The center grid cell of the object's ground truth bounding box is responsible for predicting the object. This grid cell's objectness score is one and zero for others.

[1] A tensor is a multi-dimensional array with a uniform type [26].

3.2.1 Bounding Box Prediction

Each bounding box in the original YOLO consists of four predictions: x, y, w, h. The center of a box was represented by (x,y) coordinates relative to the bounds of the grid cell. The width w and height h are predicted relative to the entire image. This approach changed in the second version of YOLO by using bounding box priors (anchors) and predicted offsets instead of coordinates. Predicting offsets instead of coordinates simplified the problem and made it easier for the network to learn.

Anchors are initialized with two prior anchor dimensions: width p_w and height p_h. The network uses these priors to predict height t_h, width t_w, and center coordinates (t_x,t_y). Figure 3.3 provides a graphical representation of the anchor-based learning problem. The following equations transform the predictions to obtain bounding boxes:

$$b_x = \sigma(t_x) + c_x \tag{3.2a}$$

$$b_y = \sigma(t_y) + c_y \tag{3.2b}$$

$$b_w = p_w \cdot e^{t_w} \tag{3.2c}$$

$$b_h = p_h \cdot e^{t_h} \tag{3.2d}$$

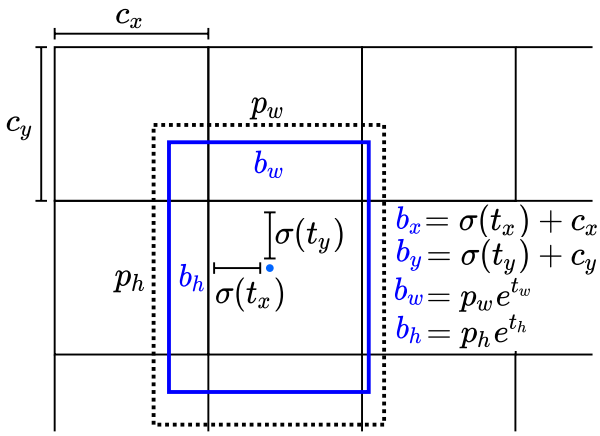


Figure 3.3: Anchor box [24]

The anchor box priors are determined by k-means clustering. The YOLO authors sort of just chose, these are their words, 9 clusters and 3 scales arbitrary and then divide up the clusters evenly across scales and boxes. On the COCO dataset, they end up with: [(10 x 13),(16 x 30),(33 x 23)],[(30 x 61),(62 x 45),(59 x 119)],[(116 x 90),(156 x 198),(373 x 326)].

3.3 Architecture

The YOLOv4 architecture is composed of three parts, a backbone for extracting features, a neck that is used for collecting feature maps from different stages, and a head that predicts classes and bounding boxes of objects. Figure 3.4 depicts the architecture. This section will describe each part separately.

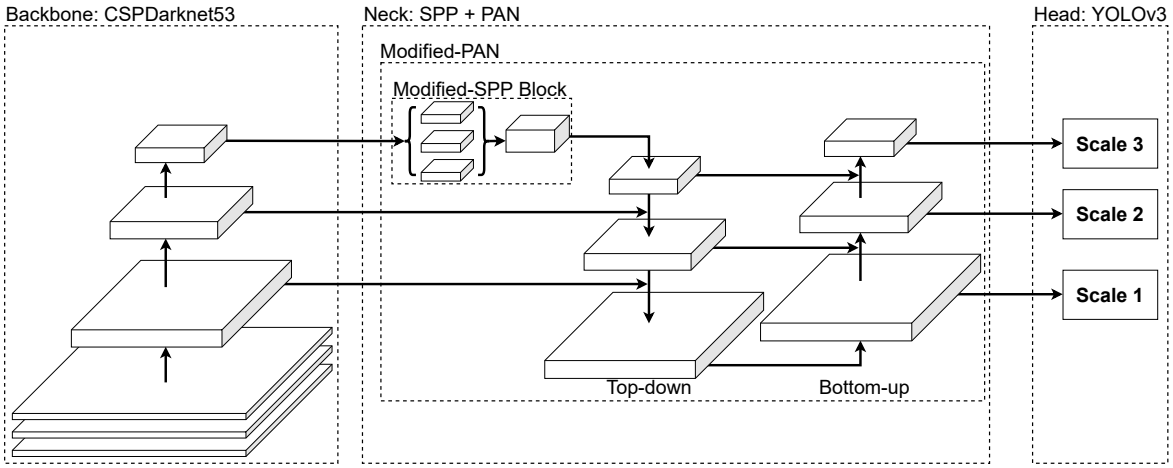


Figure 3.4: YOLOv4 architecture overview.

3.3.1 Backbone

Extracting features from the input images is the first step of the network. For this step, YOLOv4 modifies the Darknet53 CNN as used in YOLOv3. The Darknet53 network uses successive 3 x 3 and 1 x 1 convolutional layers and skip connections known as residual connections [27]. Modifying Darknet53 by implementing Cross Stage Partial (CSP) networks result in the network being used by YOLOv4: CSPDarknet53. This network consists of five CSP blocks, which in their turn use n residual blocks. Before each CSP block, the input feature map is down-sampled by a convolutional layer. Feature maps are extracted at three different stages: after the third, fourth, and fifth CSP block. A complete overview of the CSPDarknet53 is presented in Figure 3.5.

The backbone is trained separately from the entire YOLOv4 network on the ImageNet dataset. Before training, an average pooling layer, fully connected layer, and nonlinearity layer (Softmax) are added.

CSP block

A Cross Stage Partial (CSP) [28] block, blue in Figure 3.5, splits the data channels into two parts $x = [x', x'']$ and then merges x'' with the original computation performed on x'. This splitting and merging of data has multiple advantages. First, the gradient path is doubled by the split and merge strategy. Furthermore, there is a reduction in the amount of memory traffic due to only one part being processed by the original computation. The authors of YOLOv4 added additional convolutional layers to each branch and finally perform a convolution on the concatenated feature map. These so-called transition layers maximize the difference in gradient combination.

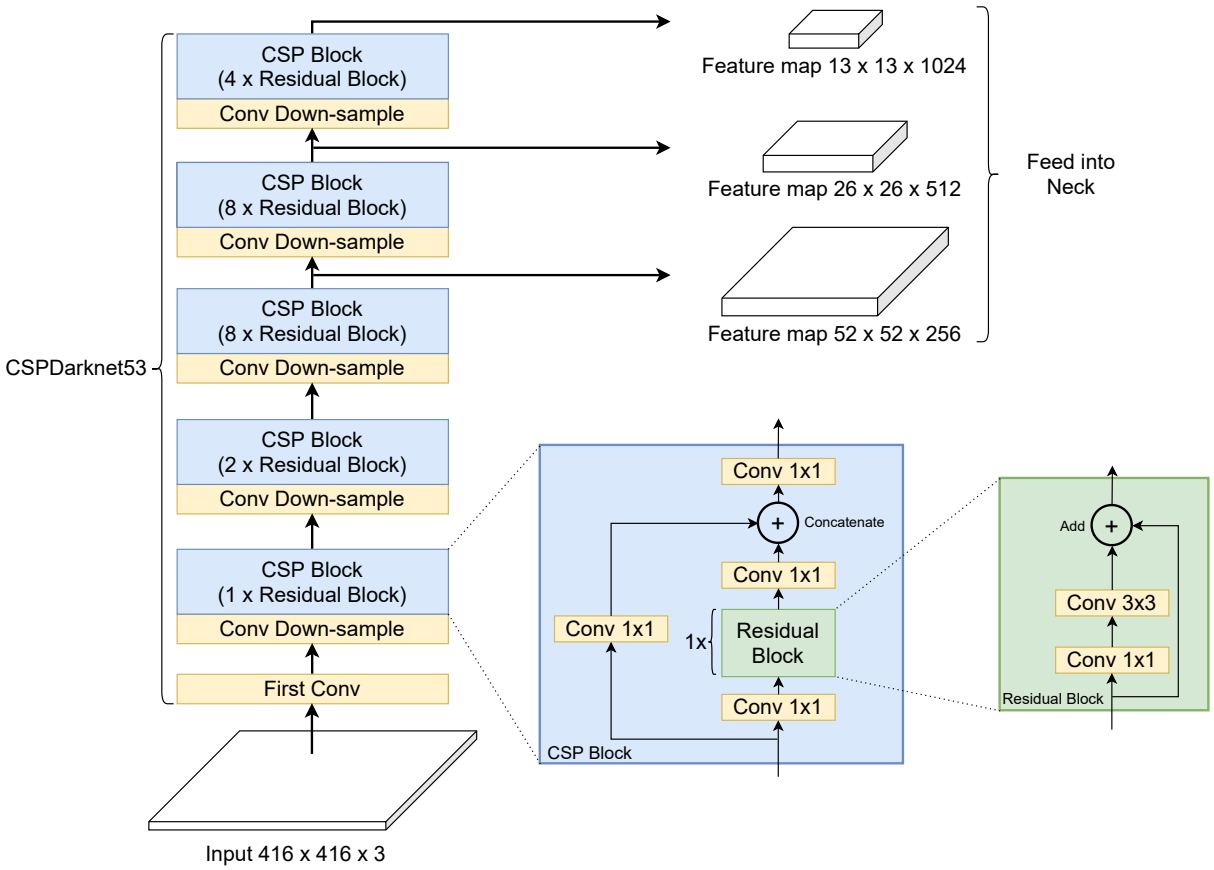


Figure 3.5: YOLOv4 backbone.

Residual block

Residual blocks [27] provide a solution for vanishing or exploding gradients in deep networks. Networks do not perform better by simply stacking more layers as shown by the inventors of the residual block. So they experimented with skip connections that perform identity mapping on their outputs. Skipping a connection is mathematically defined as $y = F(x) + x$, where x is the input (identity), y the output, and $F()$ feature mapping. This technique of identity mapping adds neither extra parameters nor computational complexity but increases the accuracy of deep networks.

The green block in Figure 3.5 represents a residual block. Feature mapping function $F()$ performs the original Darknet 3 x 3 and 1 x 1 convolution. The input is then copied to a separate branch, and both are added in the end.

3.3.2 Neck

After the backbone, there is the neck. Its goal is to enrich information feeding in from the different stages from the backbone and passing it to the head. The neck modifies and combines three different state-of-the-art methods to realise this: a Path Aggregation Network (PANet), one SPP block, and three SAM blocks. Figure 3.6 provides a graphical overview of the neck. Each block is discussed separately in this section.

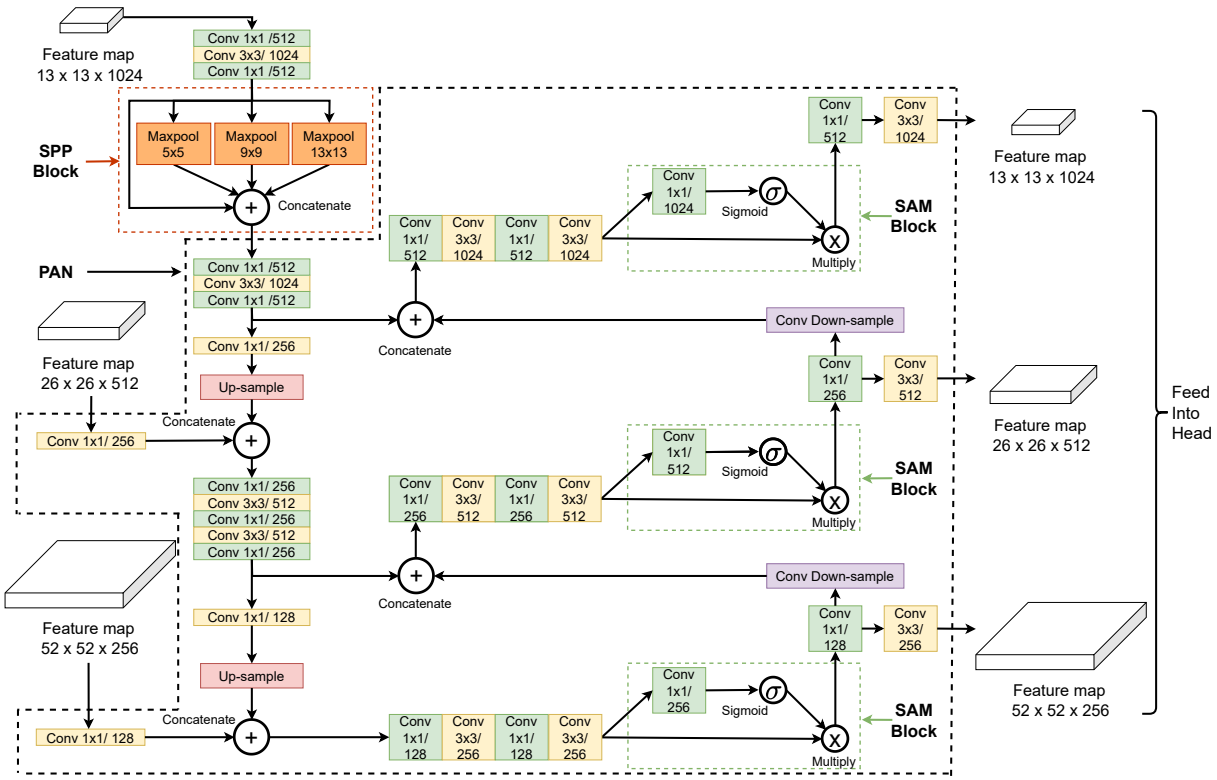


Figure 3.6: YOLOv4 neck.

PANet

The modified Path Aggregation Network (PANet) [29] starts with a bottom-up path propagating feature maps from scale three up to the first scale. This path enhances the localization capability of the entire feature hierarchy. By propagating low-level patterns such as edges or instance parts through the scales, large instances can be accurately localized and identified. This bottom-up path is identifiable in Figure 3.6 by following the stream of data flowing from low-resolution feature maps to the higher ones.

Higher-resolution feature maps respond strongly to entire objects while lower ones focus more on low-level patterns. That is why PANet implements a top-down path to propagate semantically strong features and enhance all lower resolution features.

SPP block

The modified Spatial Pyramid Pooling (SPP) [30] block performs four max-pooling operations on the input feature map with kernel sizes k x k where $k = 1, 5, 9, 13$. Note that $k = 1$ simply bypasses the other kernels as can be seen in the orange block in Figure 3.6. Each max-pooling operation receives a copy of the input, all results are concatenated increasing the dimension of

the output channel by four relative to the input. The spatial dimension is retained by applying the sliding kernel over each pixel. YOLOv4 implemented this block since it separates out the most significant context features, and significantly increases the receptive field.

SAM block

A Spatial Attention Module (SAM) [31] block improves the representation of interest, i.e., tells where to focus on. The goal of this block is to increase representation power by using an attention mechanism: focus on important features and suppress unnecessary ones. Given a feature map, the block infers attention maps along the spatial dimension. These attention maps are then multiplied to the input feature map. YOLOv4 modifies SAM from spatial-wise attention to point-wise attention. The modified SAM block is represented by the dotted green box in Figure 3.6.

3.3.3 Head

YOLOv4 deploys the same head as used in YOLOv3. Each feature map received from the neck passes through a fully connected layer implemented as a N_i x N_i x F convolutional layer with 1 x 1 filters, where $F = 3 \cdot (4 + 1 + C)$. Output F represents the 3D tensor with three boxes, N_i x N_i 1D tensors consisting of four bounding box coordinates, one objectness score, and C conditional class probabilities. Figure 3.7 depicts the head part of YOLOv4.

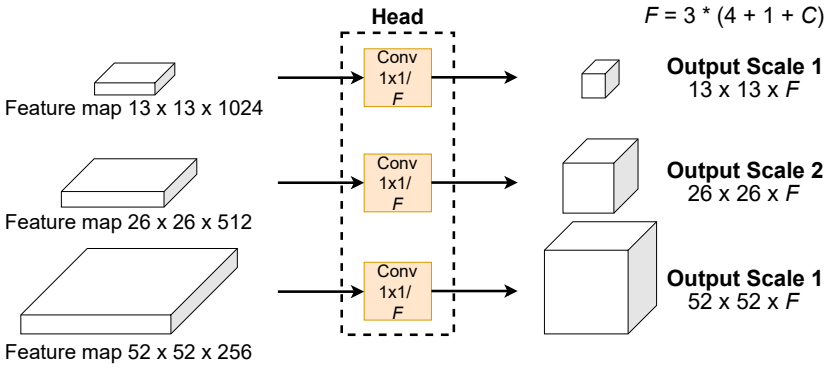


Figure 3.7: YOLOv4 head.

3.4 Processing the output

The network outputs predictions on three scales each with N_i x N_i grids. By summing all the grids, we get the total of objects that can be detected. For example, using a 416 x 416 input image, 3549 total predictions over the three scales are computed. Filtering these predictions keeping only relevant predictions is an important post-processing step. A technique often seen in literature is the non-max suppression algorithm. This algorithm is composed of two steps. First, predictions with an objectness score lower than a certain threshold are removed. Second, bounding boxes with an IoU higher or equal to a certain threshold relative to a bounding box with a higher objectness score will be discarded. Algorithm 1 represents the non-max suppression algorithm in pseudo-code.

Algorithm 1: Non-max suppression algorithm.

Input : $B = \{b_1, .., b_n\}$, $P = \{p_1, .., p_n\}$, λ_P, λ_{IoU}
B is a list of bounding boxes
P contains corresponding objectness scores
λ_P defines the objectness threshold
λ_{IoU} is the IoU threshold

Output: $B_r = \{\}$, $P_r = \{\}$
B_r is a list of non-max supressed boxes
P_r contains corresponding objectness scores

```
begin
    B_r ⟵ {}
    P_r ⟵ {}
    /* Discard all boxes with objectness under the threshold */
    for b_i in B do
        if b_i < λ_P then
            B ⟵ B − b_i
            P ⟵ P − p_i
        end
    end
    /* Discard boxes with high IoU relative to a box with a higher objectness score */
    while B ≠ empty do
        P_max ⟵ max(P)
        B_max ⟵ b_{P_max}
        B_r ⟵ B_r + B_max
        P_r ⟵ P_r + P_max
        B ⟵ B − B_max
        P ⟵ P − P_max
        for b_i in B do
            if IoU(B_max, b_i) ≥ λ_IoU then
                B ⟵ B − b_i
                P ⟵ P − p_i
            end
        end
    end
    return B_r, P_r
end
```

3.5 Related Applications

The goal of this section is to provide the reader with a short overview of applications using YOLO. First, the applications described in four different papers are elaborated. After that, a webinar from the Catapult developer Mentor Graphics is summarised, giving an idea of how a project using Catapult could be approached.

A Demonstration of FPGA-based You Only Look Once version2 (YOLOv2)

The application described in [32] uses a host PC to send images via Ethernet to an FPGA board that implements YOLOv2. After processing the image, the location and classification of detected objects are sent back. Figure 3.8 demonstrates the system diagram. A demo is available on

YouTube[2].

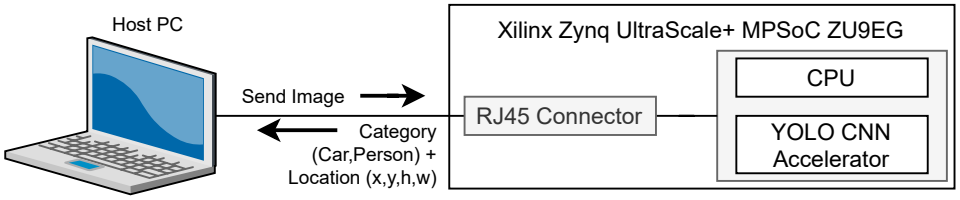


Figure 3.8: System diagram. Figure adapted from [32].

The overall hardware architecture is described in another paper [33]. The goal was to design a lightweight YOLOv2 version. They replaced the original backbone working with floating-point units with a binarized backbone, and parallel support vector regression (SVR) for localization and classification. This new design reduced the weight sizes with a factor of seven with a slight drop in accuracy of 2.17% compared to the conventional floating precision design. The architecture as implemented on the Xilinx ZCU102 board computed an image in 24.5 msec (40.81 FPS) which met their real-time requirements.

Off-chip DRAM stores all weights, and the architecture itself has weight caches. The input feature maps are stored using the on-chip memory on the FPGA. All layers are evaluated sequentially. Computation of the binarized convolution is realized with XNOR gates and DSP48 (48-Bit Accumulator/Logic Unit) blocks that compute the parallel SVRs. The ARM processor receives the result and applies post-processing.
The authors compared the performance with the NVidia Jetson TX2 embedded platform board. This board is equipped with an embedded CPU (ARM Cortex-A57) and an embedded GPU (Pascal GPU). The original YOLOv2 was used during the testing phase of the board. The Xilinx ZCU102 board used the lightweight YOLOv2 version for testing. Table 3.1 shows the results of their tests.

Table 3.1: Comparison with the NVidia Jetson TX2 board. Results from [33].

Platform	**Embedded CPU**	**Embedded GPU**	**FPGA**
Device	Quad-core ARM Cortex-A57	256-core Pascal GPU	Zynq Ultra. MPSoC
Clock Freq.	1.9 GHz	1.3 GHz	0.3 GHz
Memory	32 GB eMMC Flash	8 GB LPDDR4	31.1 Mb BRAM
Time [msec] (FPS) [sec-1)	4210.0 (0.23)	715.9 (1.48)	24.5 (40.81)
Power [W]	4.0	7.0	4.5
Effiency [FPS/W]	0.057	0.211	9.06

A Power-Efficient Optimizing Framework FPGA Accelerator Based on Winograd for YOLO

Bao [34] proposes an accelerator for YOLO using the PYNQ architecture. The accelerator is based on the Winograd algorithm that is used to improve the traditional convolution used in CNNs. This section does not go further into detail on this algorithm but focuses more on the systems architecture.

PYNQ, short for **Py**thon Productivity for Zy**nq**, is an open-source project from Xilinx that integrates a multi-core processor and an FPGA into a single integrated circuit. This allows for the

[2] https://www.youtube.com/watch?v=_iMboyu8iWc&ab_channel=HirokiNakahara

creation of high-performance embedded applications that take advantage of the FPGA fabric while using the Python language. PYNQ uses a Linux kernel with Python APIs running on top. The software application is accelerated with the use of a Programmable Logic (PL) Overlay. An Overlay is a Python wrapper around an underlying PL hardware design. This way, hardware co-processors, and peripherals are accessible as function calls. More information on using PYNQ with neural networks can be found here[3].

Figure 3.9 presents the overall system overview. On the PS side, PYNQ implements the Linux kernel on the ARM cores. The main application is running in the Python environment and communicates with the PL. Execution of the accelerator is scheduled by the CPU. The CPU stores the input feature maps in the external DDR. On the PL side, data from the external DDR is cached in the on-chip RAM to be then processed by the accelerator. The computed result is read back by the CPU via the AXI bus and executes the application of image post-processing and display.

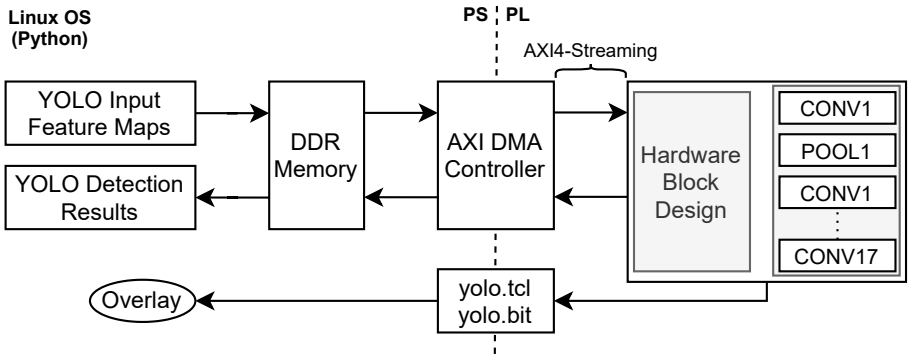


Figure 3.9: System overview. Figure adapted from [34].

An FPGA Implementation of Real-time Object Detection with a Thermal Camera

The application described in [35] processes images from a thermal camera and creates a gray-scale image with detected objects. Object detection is implemented on a Xilinx ZCU102 board using the YOLOv2 object detection neural network. Figure 3.10 shows the overall system overview. The thermal camera captures a four-channel image (RGB and thermal) and sends them to the laptop PC. These images are then resized to the correct format and fed into the ZCU101 board through an Ethernet cable. The laptop PC receives the computed result and applies post-processing to compute the output image. Instead of using a thermal camera, the CAMEL dataset provides thermal and RGB images for objection detection and tracking.

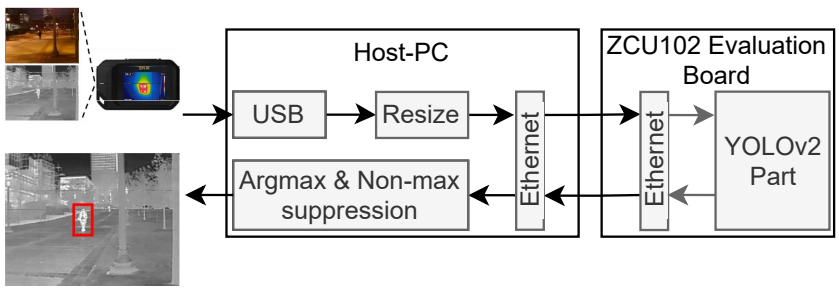


Figure 3.10: System overview. Figure adapted from [35]

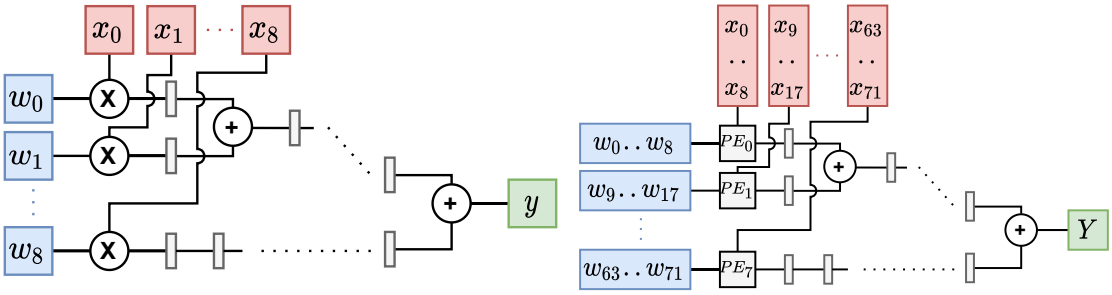
.

[3] https://connect.linaro.org/resources/san19/san19-313/

Accelerating Tiny YOLO v3 using FPGA-based Hardware/Software Co-Design

The developers of [36] developed an FPGA-based accelerator to speed up the YOLOv3 tiny model. The Xilinx Virtex 7 VC707 FPGA was used. The first step was to design the model in Python using TensorFlow. Profiling shows that convolutions are by far the most complex and time-consuming operation of the model.

In the second step, weights are extracted, and the model is implemented in ANSI-C since the design will be synthesized using the Vivado High-Level Synthesis tooling. The building block of the accelerator is the processing element PE_t consisting of 3x3 multiply-adds (MultAdds) representing the maximum dimensions of a filter. The PE instantiates 9 parallel DSPs computing the multiplication in a single clock cycle, see Figure 3.11a. An adder tree accumulates all results and utilizes pipeline registers to generate one output per clock cycle.



(a) PE_t (b) PE_v

Figure 3.11: Accelerator architecture.

The PE_t is then eight times instantiated in a volume-based processing element PE_v. This PE has the same architecture but replaces the multiplications of PE_t with PE_t, as can be seen in Figure 3.11b. The total computation that can be done in parallel now equals 3x3x8 multiplications. Finally, a top-level module, see Figure 3.12, is created that instantiates 32 PE_v blocks.

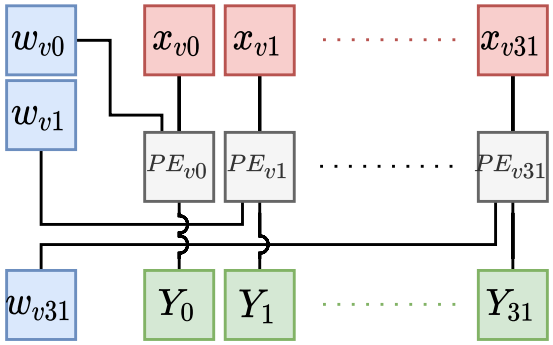


Figure 3.12: Top-level module

Figure 3.13 shows the complete system overview with the PL containing the accelerator block speeding up convolutional layers. All other computations are done in the PS. Inputs and filters are stored in DRAM and can be fetched over the AXI interconnect by the accelerator. The PL controller inside of the accelerator fetches data from DRAM and stores it into the local cache, i.e., input buffer or filter buffer. Computed results are sent back to DRAM via the controller.

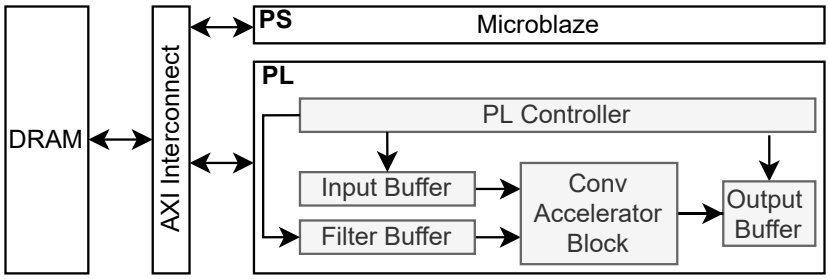


Figure 3.13: System overview. Figure adapter from [36]

From HLS Component to a Working Design

The From HLS Component to a Working design webinar [37] from Mentor Graphics talks about taking a HLS component and putting it into a context of a larger design specifically in terms of hardware and its software interfaces and verifying it within the context of the system. The designed application in this talk is based on YOLO tiny and is further described in a corresponding manual [38]. Figure 3.14 shows the oversimplified system implementing the application. Images are taken from the webcam and processed by the CPU accelerated by the machine learning accelerator, the final result is displayed on a monitor.

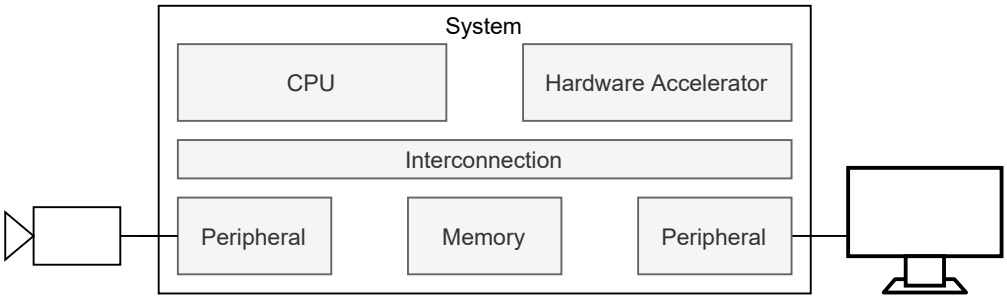


Figure 3.14: Oversimplified system overview.

The design is divided into five steps: (1) host execution, (2) host and Catapult-C, (3) TLM + Catapult-C, (4) TLM + RTL, (5) Full RTL. Each step is described in detail below.

The first step begins with implementing and running all components as shown in Figure 3.14 on a host computer for algorithmic verification. The TensorFlow framework is used to implement the YOLO tiny model. In this step, Python executes the complete application.

Now that the system correctly works on the host computer, the next step is to convert Python code to Catapult compatible C code. Figure 3.15 illustrates this step by converting each underlying function of a neural network layer. These replacement C functions are then plugged back into the original Python code to verify its correctness. Partitioning the algorithm is an important step in this process. For example, the pre-processing of images for scaling the pixel values and resizing the image is done in software. But implementing the object detection in C makes more sense because there are only data dependencies between neural network layers.

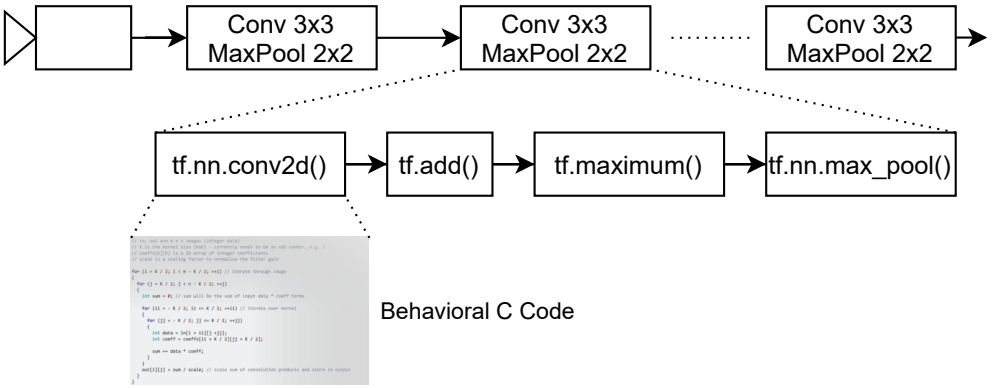


Figure 3.15: Convert Python code to behavioral C. Figure adapter from [37].

Another important process within step 2 is applying algorithm modifications such as defining memory architecture, loop unrolling, pipelining, floating-point to fixed-point conversion, and reduced precision. Research on the ResNet deep neural network showed that reducing the 32-bits weights to 8-bit weights only affects the accuracy by less than 0.1%. This reduces the YOLO tiny weights from 34 MB to 8.5 MB. Another important impact is the fact that an 8-bit multiplier is about 1/16th the area of 32-bit multipliers, thereby saving area and energy. The bus bandwidth was also considered since the 8-bit 3x3 convolutional kernel required two cycles on the 64-bit bus. By reducing the weights to 7-bit, it only takes one bus cycle to transfer the complete kernel.

In step three, the function call interfaces are replaced by a transaction interface. The HLS component can interface in multiple ways to the rest of the system. These interfaces can be easily added through the Catapult tool. To model the components in a more realistic way, a virtual prototype is made. This prototype is an abstract model of the design modeled at the transaction level (TLM) in a language such as SystemC or System Verilog. This model allows for the verification of cross-compiled code, drivers, and interfaces between hardware and software. The CPU, interconnect and memory are modelled using the TLM environment using SystemC, the accelerator is made in C and the peripherals are still managed by the host PC.

The last two steps convert all components to RTL and perform performance and power analysis. Finally, the design is loaded onto a FPGA development board.

4 CATAPULT HIGH-LEVEL SYNTHESIS

Catapult is a high-level synthesis (HLS) tool developed by Mentor Graphics that creates RTL implementations from compatible C, C++, and SystemC design specifications. Designers use C, C++, or SystemC to describe the structure and behaviour of the design. The description is written in such a way that Catapult can synthesize the interfaces, data structures, and loops to a specified FPGA or ASIC technology and produce an optimized RTL implementation [7].

The complete HLS design flow comprises multiple tools and steps, as illustrated in Figure 4.1. Designers first implemented and test the design specification in C, C++, or SystemC. Then, HLS is run, highlighted in red, on the source code together with technology and clock information to generate RTL code. The last step is to synthesize the RTL code from Catapult using Xilinx Vivado. Design integration takes place at this step, the Catapult output may then be merged with other IP/RTL blocks.

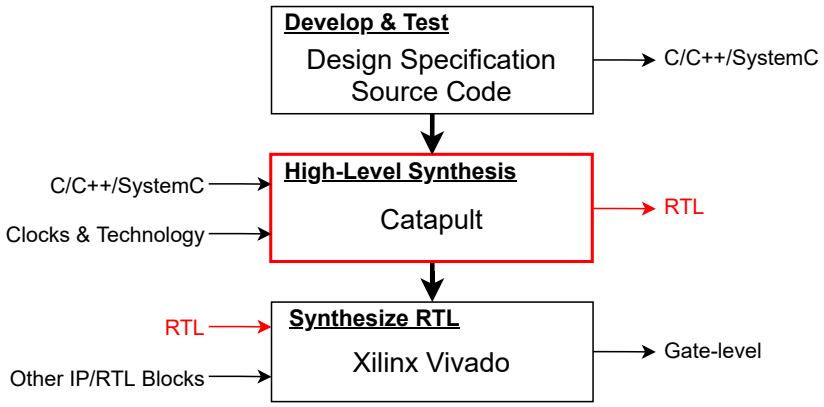


Figure 4.1: High Level Synthesis design flow [7]

This chapter first explains the first step of the design flow in which the basics for creating a compatible HLS C++ design for Catapult are elaborated. Then, the Catapult workflow is presented and its verification tools are explained. Later in the report, in Section 6.7, the Catapult HLS workflow for synthesizing the design made in this work is explained. Section 6.8 then continues on how the Catapult generated RTL is integrated into the complete design and synthesized using Xilinx Vivado.

4.1 Data types

Mentor Graphics developed bit-accurate data types for use in HLS known as Algorithmic C data types. Building the design in C++ using Algorithmic C data types results in the hardware behaviour exactly matching the described behaviour. Algorithmic C supports integer and fixed-point data types. All types have support for the standard C++ arithmetic and logical operators.

4.1.1 Integer Data Types

Integer data types model a signed or unsigned bit vector with static bit precision. Integers can be defined after including the ac_int.h header. Algorithmic C integers are templatized, this allows for a configurable width and signedness of variables:

```
#include <ac_int.h>
ac_int<W,false> x; //Unsigned Integer
ac_int<W,true> x; //Signed Integer
```

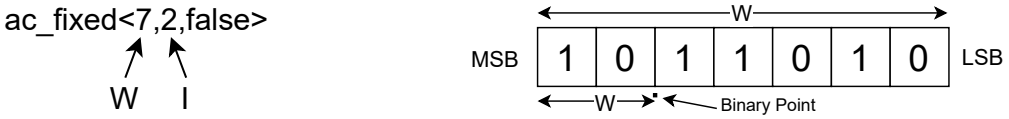
Parameter W determines the bit width and the boolean if the integer is signed.

4.1.2 Fixed Point Data Types

Algorithmic C fixed-point data types model a signed or unsigned bit vector with static fixed point precision. Fixed point variables can be declared after including the ac_fixed.h header. Fixed point variables are declared as:

```
#include <ac_fixed.h>
ac_fixed<W,I,false> x; //Unsigned Fixed Point
ac_fixed<W,I,true> x; //Signed Fixed Point
```

The functionality of parameters W and the boolean correspond to that of integer data types, but the I parameter determines the location of the decimal point relative to the MSB as shown in Figure 4.2.



(a) Definition (b) Bit representation

Figure 4.2: Fixed Point data types example.

Fixed-point data types have support for quantization, truncation, and saturation. These techniques are not further described since fixed-point data types are not used in this work.

4.2 Slice

Algorithmic C data types support reading a slice from the original variable using the slice method:

```
slc<W>(int lsb)
```

Parameter W determines the width of the slice and lsb points to where the slice begins. An example is provided in Figure 4.3 with the corresponding code:

```
ac_int<8,false> config = 10;
ac_int<4<false> filter_id;
filter_id = config.slc<4>(1);
```

A slice of a variable can be set using the set_slc method:

```
set_slc(int lsb, const ac_int<W,S> &slc)
```

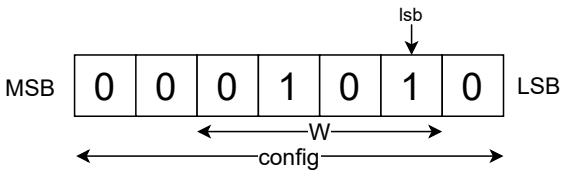


Figure 4.3: Algorithmic C slc method example.

4.3 Block Design

Catapult allows blocks to be designed in either a function-based or class-based way. This work only uses class-based block design because of the easy extensibility that class-based blocks provide. As a result, no further information on function-based block design will be provided. Class functions can be configured to be either *Top*, *Block*, or *Inline*. Exactly one function in the complete design must be designated *Top*, meaning the top-level block for the entire design. Functions configured as *Block* result in sub-blocks. The *Inline* setting moves a function inside one of the blocks.

The code below illustrates how class-based designed blocks are implemented by means of an accumulate class. Figure 4.4 shows the hardware implementation of the Accumulate class created after synthesis. Variables defined under the private label are considered static and synthesized as memory elements. So, on line 4 for example, the acc variable will be synthesized as a 32-bit unsigned integer memory element. Catapult determines the reset value of static variables by either the value assigned to a variable under the private label or the value assigned in the constructor.

```
1  class Accumulate
2  {
3  private:
4   ac_int<32,false> acc = 0;
5  public:
6    Accumulate(){}
7  #pragma hls_design top
8    void run(ac_int<8,false> din[4], ac_int<32,false> &dout){
9  #pragma hls_unroll no
10     ACCUM: for (int i=0; i<4; i++) {
11       acc += (ac_int<32,false>)din[i];
12     }
13     dout = acc;
14    }
15  };
```

The run function is designated as the top-level function on line 7. This function uses a for-loop to accumulate four 8-bit unsigned integers to the acc variable. Loops can be labeled, i.e., ACCUM in this example, so that Catapult can identify and analyze them allowing them to be unrolled or pipelined. The loop is left rolled as defined on line 9, note that this can be changed manually in Catapult. Finally, on line 13, the result is sent out. Catapult also generates a corresponding schedule, which, in this case, is composed of 4 states/cycles.

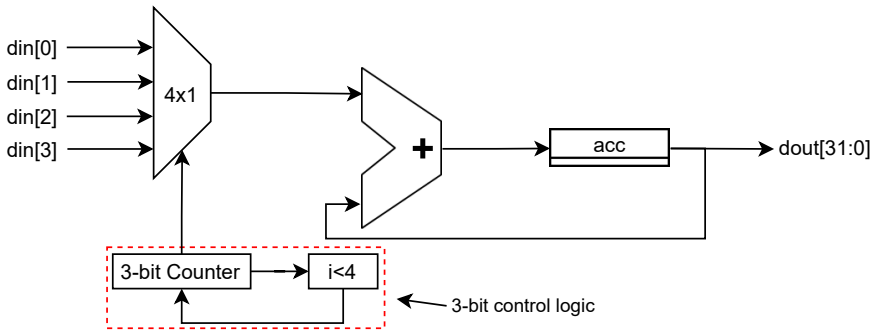


Figure 4.4: Hardware implementation of the accum class [7]

4.4 I/O

There are two ways for passing IO into and out of a design, pass by reference or pass by value:

```
void run(ac_int<8,false> &din, ac_int<32,false> &dout) //Pass by Reference
void run(ac_int<8,false> din, ac_int<32,false> dout)   //Pass by Value
```

Each way leads to different behaviour. Pass by reference is when a variable is declared as a reference. Variables declared as reference are stored externally, i.e., data is stored off-chip. Catapult allows mapping these variables to off-chip storage in either registers or memory. When mapped to DRAM, for example, Catapult synthesizes an additional bus interface and logic that handles data transactions.

Variables passed as reference are fetched each time they are accessed in the code. Pass by value variables, however, fetches the data and registers the data internally in the design. This has the benefit of IO data not having to be held stable after it is read and it reduces IO traffic.

Both methods can either be implemented as conditional or unconditional IO. Conditional IO synchronizes the transfer of data via ready/acknowledge control signals. Unconditional IO is simply mapped to a wire type resource, i.e., without handshaking protocols. Configuring IO to be either conditional or unconditional is done at design time in Catapult.

4.5 Hierarchical Design

Hierarchy can be added to the design by partitioning the design into several blocks. This allows blocks to run in parallel resulting in higher throughput. Another reason for hierarchy is that blocks running at different transfer rates can be connected. Hierarchical blocks can be pipelined.

Class-based hierarchical blocks can only have one hierarchical function, named *run* in this work. Only one hierarchical function is allowed to be called, otherwise, the system could not pipelined. Hierarchical blocks must be labeled with *#pragma hls_design interface* to be detected by Catapult. The top hierarchical block adds *top* to this pragma. Non-hierarchical blocks may be used multiple times in different hierarchical blocks. Figure 4.5 shows an example of a calling tree for a hierarchical design together with non-hierarchical blocks. In this example, blocks 1 to 4 are synthesized into separate blocks that run in parallel while the non-hierarchical are inlined. Inlining results in, for example, *Function B* being instantiated twice essentially.

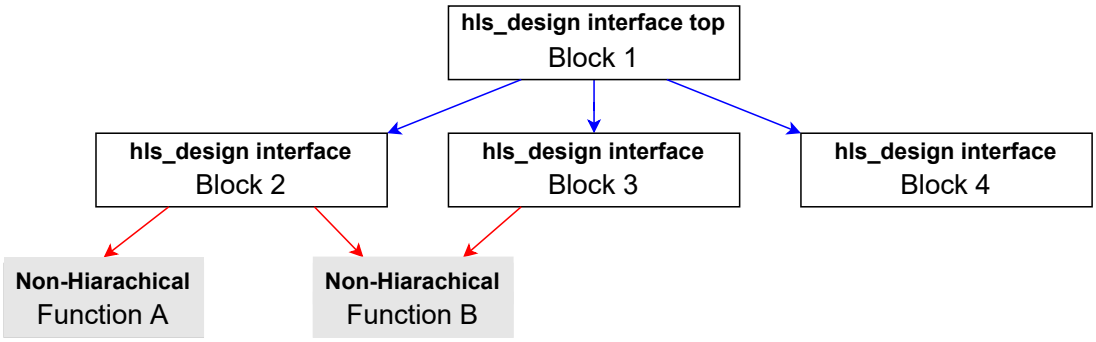


Figure 4.5: Hierarchical calling tree [7]

4.5.1 Algorithmic C Channel Class

Hierarchical blocks exchanging data such as variables or arrays require Catapult to automatically insert synchronization. The Algorithmic C ac_channel class library allows modelling these constructs. The ac_channel class implements essentially a C++ FIFO with a ready/valid handshake protocol that guarantees that the reading and writing of data between blocks occurs in the same order. An ac_channel is defined as follows:

```
#include <ac_channel.h>
ac_channel<T > my_channel;
ac_channel<T > my_channel(<prefil_number>, <prefill vallue>) //Preloading
ac_channel<ac_int<8,false> > my_channel;
```

Parameter T can be any native, Algorithmic C, SystemC, or user-defined data type. A channel may be pre-loaded, this can, for example, be used for feedback channels. Channels support writing and reading data. For one channel there can only be one block writing (producer) and one reading (consumer). Before synthesizing, the FIFO depth of a channel has to be set in Catapult. Writing to a full channel during C++ simulation results in an assert. The synthesized hardware will block when attempting to write a full FIFO. An example of writing to a channel:

```
ac_int<8,false> tmp = 10;
my_channel.write(tmp);
```

Reading data is implemented with the read method. The C++ simulation asserts when reading an empty channel. The synthesized hardware will block when attempting to read an empty FIFO. An example of reading data from a channel:

```
tmp = my_channel.read()
```

To prevent an assertion during simulating the C++ design, a check is performed to verify that data is present before reading. This check is implemented using the available method. This method is always synthesized to true, making it only useful for C++ simulation:

```
if (my_channel.available())
    tmp = my_channel.read()
```

The limitation of the read and write methods is the potential for stalling the hardware. This forms a problem if it were to be essential to do something else if the data is not available. This is solved by the non-blocking size method which reads the channel size and can then check if data is present to read:

```
bool available = input.size()>0;
if (available)
    tmp = my_channel.read()
```

4.5.2 Example

This section provides an example of how two hierarchical sub-blocks are connected via a top hierarchical block. Note that top hierarchical blocks can only implement interconnects but no logic. The first sub-block (Modulo) reads the value of din and applies modulo of 10 to it. The result is then written to the interconnecting channel:

```
//Block 1: Class Modulo. Class attributes and constructor omited
#pragma hls_design interface
void run(ac_int<8,false> &din, ac_channel<ac_int<8,false> > &dout){
  ac_int<8,false> tmp = din % 10;
  dout.write(tmp);
}
```

Block 2 (Accumulate) first checks if data is present in the interconnecting channel. If data is present, it is accumulated and sent out via the dout output:

```
//Block 2: Class Accumulate. Class attributes and constructor omited
#pragma hls_design interface
void run(ac_channel<ac_int<8,false> > &din, ac_int<32,false> &dout){
  if (din.available()){
    acc += (ac_int<32,false>)din.read();
  }
  dout = acc;
}
```

Both blocks are connected via the *Top* class:

```
1  class Top
2  {
3  private:
4    ac_channel<ac_int<8,false> > interconnect(1,0); //Preload
5    Modulo mod;
6    Accumulate acc;
7  public:
8    Top(){}
9  #pragma hls_design interface top
10   void run(ac_int<8,false> &din, ac_int<32,false> &dout){
11     mod(din, interconnect);
12     acc(interconnect, dout);
13   }
14 };
```

4.6 Workflow

The Catapult workflow comprises the synthesis of the C++ design and test/verification tools that verify the correctness of the design files and generated RTL. The tools are described in the *Catapult Synthesis User and Reference Manual* [7]. This section is about the verification tools

since the actual synthesis task is elaborated later in this work when synthesizing the final design. Figure 4.6 illustrates the Catapult workflow. Catapult independent files/tasks are represented by white boxes and all other boxes by Catapult tools. Unfortunately, due to a lack of time, only the SCVerify tool is used for verification.

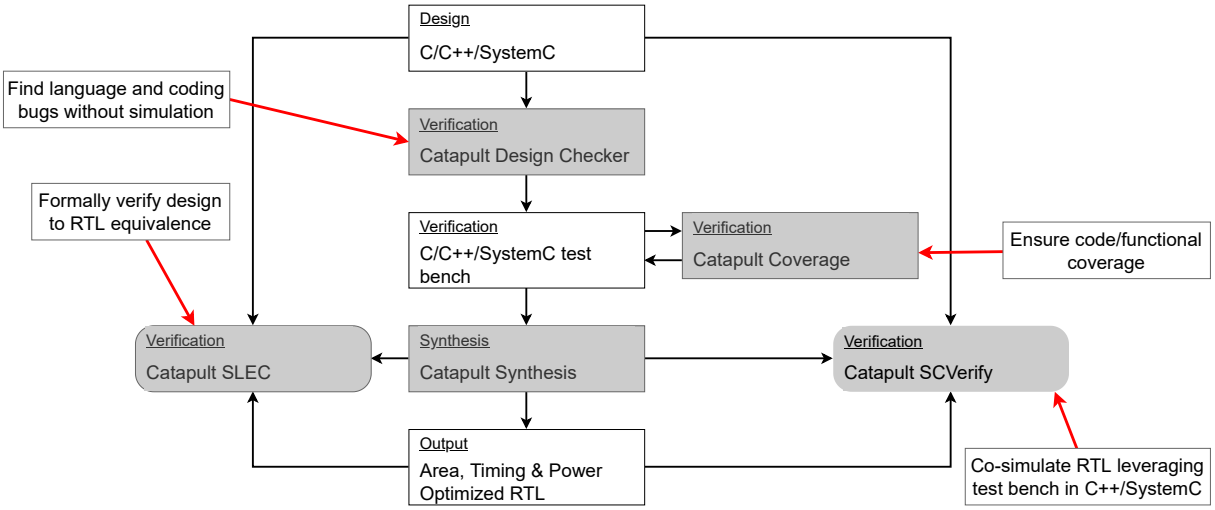


Figure 4.6: Catapult workflow.

4.6.1 Catapult Design Checker

First, the design is statically checked using the Design Checker that helps identify ambiguous behaviour. It checks if the proper HLS coding practices are followed and if other coding errors are avoided, such as divide by zero, uninitialized memory read, overflow/underflow, etc.

4.6.2 Catapult Coverage

Catapult Coverage (CCOV) is a hardware-aware tool that takes synthesis intent into account when calculating the code and or functional coverage of the C/C++/SystemC test bench test vectors.

4.6.3 Catapult SLEC

The Catapult Sequential Logic Equivalence Checking (SLEC) tool provides a way to check the functional equivalence between the C++ design and generated RTL by Catapult.

4.6.4 Catapult SCVerify

After synthesizing the design, SCVerify can verify the RTL netlist against the original C++ design using the C++ test bench. SCVerify generates wrappers, synchronization signals, and makefiles to compile and simulate both designs and automatically compare the outputs for differences. The C++ design is considered the golden model for verification. Figure 4.7 illustrates what the SCVerify test bench looks like. The test bench is either completely in VHDL or Verilog depending on which RTL type is tested.

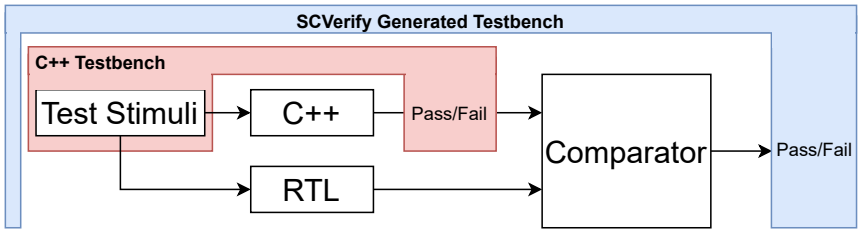


Figure 4.7: Catapult SCVerify structure [7]

By double clicking on the makefile, Catapult opens Questasim and sets up the test bench. After starting the simulation with *run -all*, the SCVerify test bench starts executing. Feedback is provided on the result of a test.

5 PROBLEM ANALYSIS

The hardware accelerator to be designed for the YOLOv4 algorithm is tightly coupled with a processing system (PS). This chapter first describes how the PS is used and how the software running on it is implemented in Section 5.1. After implementing the software and running YOLOv4 on the PS, computationally intensive functions are identified in Section 5.2. Finally, Section 5.3 analyzes the computationally intensive function to understand how hardware accelerating it should be realized.

5.1 Software Implementation

The development is targeted on the ZedBoard which integrates a dual-core ARM Cortex-A9 based PS running at a maximum frequency of 667 MHz. Since this project focuses on the hardware accelerator, only one core running a bare-metal software implementation is used to reduce PS development complexity. To further reduce software complexity, a DNN framework providing the implementation of DNN layers and support for inferring models is used.

This section will first present the implementation of the software application using the DNN framework running on the PS. Then, the workflow created in this project for the development and deployment of the YOLOv4 model is described. And finally, the interface implementation realizing data exchange between the ZedBoard and the Host PC is explained.

5.1.1 DNN Framework

Deciding which DNN framework to use is important because this both affects the PS and accelerator. Therefore, the frameworks in Section 2.3 were compared and it was decided to go for the TensorFlow [22] framework. TensorFlow has, compared to the other frameworks, the best tooling, documentation, and support. Although TensorFlow is written in C++ and allows for easy use via a Python API, it is not compatible with the PS since the code is optimized for CPU and GPU with Operating System (OS) support.

Another framework, developed by TensorFlow, called TensorFlow Lite (TF-Lite), opposed to TensorFlow, supports embedded devices. TF-Lite comprises of tools to run a TensorFlow model on embedded devices. The tool used in this project is called the TF-Lite converter, which converts 32-bit floating-point weights and operations of a TensorFlow model to 8-bit integers. This conversion has, according to TensorFlow [39], the benefit of a four-time reduction in weight size and a 3x+ speedup on microcontrollers. Reducing precision of both weights and operations leads to a decrease in accuracy. According to TensorFlow [40], the accuracy of the previous YOLO model (YOLOv3) dropped by 2.43% from 0.577 mAP to 0.563 mAP after conversion. Another benefit of TF-Lite is that the code is optimized for the ARM Cortex-A series with the use of SIMD instructions.

Although TF-Lite seems to be a good fit for this project, it requires an OS to operate, which is not available in the bare-metal software implementation. To address this issue, TensorFlow created the TensorFlow Lite Micro (TFLM) framework [41], which does not require OS support and any standard C or C++ libraries. TFLM comes with implementations of layers (kernels) optimized for the ARM Cortex-M series but allows developers to create custom kernels. Since the PS consists of a Cortex-A series CPU, kernels optimized for the Cortex-M series originally shipped with TFLM have been replaced with the TF-Lite Cortex-A series optimized kernels.

Implementation

The bare-metal software application makes use of the TFLM framework. This section summarizes the process of loading a model down to retrieving the output of a DNN. The first step is to instantiate the model from a char array, represented as my_model:

```
const tflite::Model* model = tflite::GetModel(my_model);
```

More information on this char array containing the model can be found in Section 5.1.2.

To make the final binary as small as possible, only the required kernels are loaded. The kernels required for YOLOv4 are added via the resolver object:

```
static tflite::MicroMutableOpResolver<10> resolver;
resolver.AddAdd();
resolver.AddLogistic();
resolver.AddMul();
resolver.AddConv2D();
resolver.AddPrelu();
resolver.AddPad();
resolver.AddQuantize();
resolver.AddConcatenation();
resolver.AddMaxPool2D();
resolver.AddResizeNearestNeighbor();
```

Since TFLM assumes that dynamic memory allocation is unavailable, a contiguous memory area needs to be supplied, known as arena. This arena holds intermediate results and other variables the interpreter needs:

```
const int tensor_arena_size = 13844 * 1024;
uint8_t tensor_arena[tensor_arena_size];
```

The fourth step is to create an interpreter instance. An error reporter instance may be passed into the interpreter, which allows it to write logs. Another additional object that can be passed is a profiler:

```
tflite::MicroErrorReporter error_reporter;
tflite::MicroProfiler profiler;
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena, tensor_arena_size,
     &error_reporter, &profiler);
```

Next, the interpreter allocates memory from the arena for the chosen kernels, such as memory locations, to store inputs and outputs:

```
interpreter->AllocateTensors();
```

Now that the interpreter is set up, an input can be provided. The input, as shown in the following code, loads an image into the input of the model:

```
TfLiteTensor* input = interpreter->input(0);
for (int i=0; i<img_len;i++){
    input->data.int8[i] = img[i];
}
```

Running the model is performed by calling Invoke() on the interpreter:

```
interpreter->Invoke();
```

Finally, the output is obtained as follows:

```
interpreter->output(0);
```

5.1.2 Workflow

The workflow created for this project to develop and deploy a DNN model on a single Cortex-A9 of the ZedBoard is illustrated in Figure 5.1. YOLOv4 was originally developed for the Darknet framework so the online available pre-trained weights are stored in the Darknet format. This format differs from that of TensorFlow. Converting these weights to compatible TensorFlow weights is realized by a custom weight converter that uses the Darknet weights and the YOLOv4 model made in TensorFlow to produce compatible TensorFlow weights.

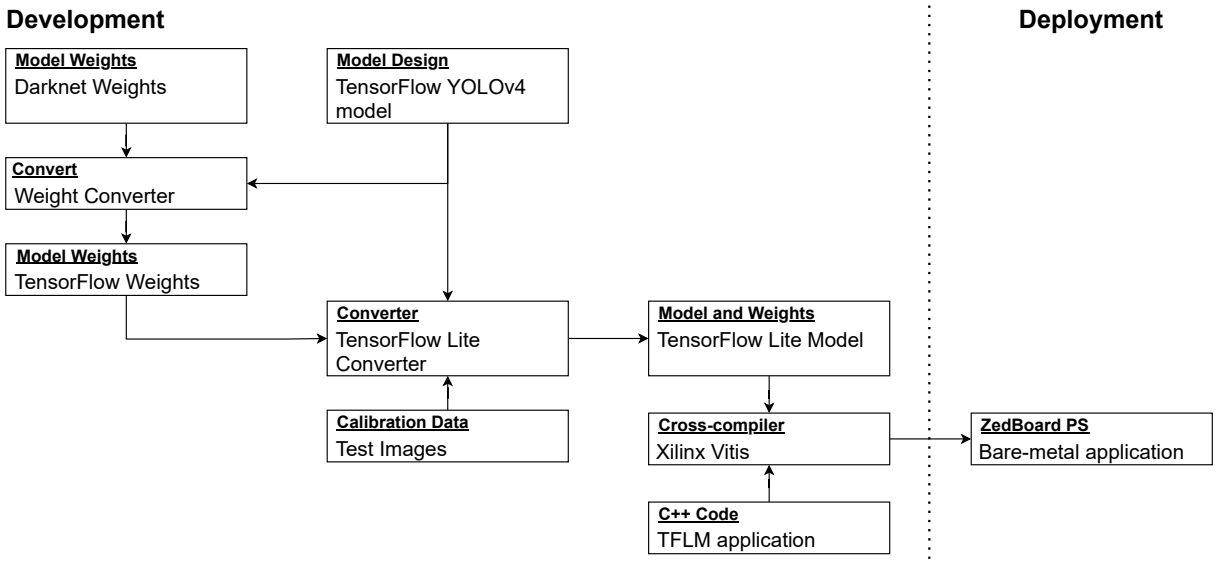


Figure 5.1: TensorFlow Lite Micro custom development and deployment workflow.

Now that the model and weights are both in TensorFlow format, conversion to TF-Lite can start. The TF-Lite Converter takes the model, weights, and a set of test images, and creates a TF-Lite model. The converter applies full integer quantization, quantizing 32-bit floating-point weights to 8-bit weights. For quantization, the calibration or range estimations take place, which finds

the minimum and maximum of all floating-point tensors in the model. Therefore, a small subset of around 100-500 samples [39] is required. See Section 5.3.1 for more information on the quantization scheme. The TF-Lite Model is stored as a char array which contains both the weights and a description of the model:

```
const unsigned char my_model[] = {
 0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
 //Lines omitted
 };
const int my_model_len = 9563288;
```

After generating the TF-Lite model, it is cross-compiled together with the TFLM application as described in the previous section. The Vitis software development platform from Xilinx takes care of cross-compilation. Finally, the cross-compiled binary is sent to the ZedBoard and the Cortex-A9 core starts executing.

5.1.3 Interface

The YOLOv4 algorithm does all of its computations on the ZedBoard but all other computations such are pre- and post-processing are taken care of by the Host PC. A common interface is needed to send the pre-processed image from the Host PC to the ZedBoard and retrieve the predictions produced by the ZedBoard for post-processing. This interface is realized by an Ethernet connection using the UDP protocol. Because of the limited time of this project, the YOLOv4 execution time is measured on the ZedBoard between inference cycles. This means that only the basis for data exchange is implemented resulting in possible packages being dropped. To prevent package drop, sending data is scheduled at a lower rate than should be required for real-time performance.

ZedBoard Implementation

The Xilinx Software Development Kit provides an open-source TCP/IP networking stack called LightWeight IP (lwIP). It supports protocols such as TCP, DHCP, and UDP. This section summarizes how the interface software is implemented on the PS of the ZedBoard.

The first step is to initialize all lwIP structures:

```
lwip_init();
```

After initializing lwIP, the mandatory IP addresses, Host PC port, and a MAC address for communication are defined:

```
u16_t port_hostPc = 5000;
ip_addr_t ipaddr, netmask, gateway, ipaddr_hostPc;
IP4_ADDR(&ipaddr,  192, 168, 100, 10);
IP4_ADDR(&netmask, 255, 255, 255,  0);
IP4_ADDR(&gateway, 192, 168, 100,  1);
IP4_ADDR(&ipaddr_hostPc,  192, 168, 100, 11);

unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };
```

The third step adds a network interface and initializes the Ethernet MAC peripheral onboard of the ZedBoard using its base address (PLATFORM_EMAC_BASEADDR):

```
struct netif *netif;
xemac_add(netif, &ipaddr, &netmask, &gateway, mac_ethernet_address,
    PLATFORM_EMAC_BASEADDR));
```

After adding the network interface, it is set as the default network interface for receiving traffic:

```
netif_set_default(netif);
```

Step five is to create and initialize a UDP structure that stores and describes its UDP address and port for receiving traffic:

```
udp = udp_new();
udp_bind(udp, &ipaddr, port);
```

The last step of the initialization process is to set a callback function for when a packet is received and bring up the interface which makes it available for processing traffic:

```
udp_recv(udp, recv_callback, NULL);
netif_set_up(netif);
```

lwIP sends a packet with the use of a pbuf structure. This structure may point to another pbuf structure if the data does not fit within a single packet via the .next parameter. The number of total pbuf structures in a pointer chain is indicated by .tot_len. Data is coupled to the .payload parameter and the total length of the data is indicated by .len:

```
int data[] = {1,2,3};
pbuf udp_data;
udp_data.next = NULL;
udp_data.payload = data;
udp_data.tot_len = 1;
udp_data.len = 3;

udp_sendto(udp, &udp_data, &ipaddr_host, port_host);
```

5.2 Profiling

By profiling the bare-metal application running the YOLOv4 model, computationally intensive functions are identified. TFLM comes with a built-in profiler that measures the executing time of kernels by starting or resetting a timer before executing a kernel and retrieving the elapsed time right after it is done. The profiler implementation is adapted to make use of the Global 64-bit timer on-board the PS.

TFLM by default uses unoptimized kernels for running on the Cortex-A9. These unoptimized kernels are further referred to as naive kernels. As discussed earlier, some kernels are replaced with optimized TF-Lite kernels that use ARM Neon instructions. ARM Neon is a SIMD architecture extension for the Cortex-A9. Neon instructions allow up to 16 8-bit operations [42].

Table 5.1 provides the profiling results of profiling YOLOv4 on one Cortex-A9 core running at 667 MHz with caching enabled. The total execution time of the naive kernels is given in the *Naive Time* column. For optimized kernels, the total time is presented in the *Optimized Time* column. Measuring the speed up between the naive time and optimized time is presented in the *Speed Up* column and proves that Neon instruction indeed causes a speedup. All optimized kernels stay under the maximum speedup of 16 but MAX_POOL_2D has a higher speedup. This has to do with the tiling of data which maintains locality of reference preventing some recalculations.

The *Called* column shows the number of times the kernel is used in a single inference cycle. Finally, the last and most important column *Percentage* gives the percentage of the execution time of a kernel relative to the total execution time of all kernels.

Table 5.1: YOLOv4 profiling results measured on one Cortex-A9 core running at 667 MHz.

Kernel	**Naive Time (ms)**	**Optimized Time (ms)**	**Speed Up (x times)**	**Called**	**Percentage (Optimized)**
CONV_2D	174222.85	-	-	113	99.67334
PRELU	493.06	-	-	107	0.29148
ADD	129.83	17.04	7.619	23	0.00975
LOGISTIC	82.90	15.90	5.214	3	0.00910
QUANTIZE	27.47	4.18	6.572	10	0.00239
PAD	19.91	-	-	7	0.01125
MAX_POOL_2D	9.63	0.46	20.900	3	0.00026
MUL	9.00	2.81	6.569	3	0.00078
CONCATENATION	2.82	-	-	10	0.00161
RESIZE_NEAREST_NEIGHBOR	0.06	-	-	2	0.00003

After analyzing the profiling results as presented in Table 5.1, it can be concluded that the most computationally intensive function taking 99.67% of the total execution time is the CONV_2D kernel. This kernel will be hardware accelerated and is described in more detail in the next chapter. In the next section, Section 5.3, the CONV_2D kernel is analyzed to understand what the exact behaviour of the accelerator should be.

5.3 2D Convolution Kernel Analysis

This section analyzes the CONV_2D kernel as implemented in TFLM. First, the quantization scheme is elaborated in Section 5.3.1. Section 5.3.2 presents the algorithm of the CONV_2D kernel.

5.3.1 Quantization Scheme

TFLM uses a quantization scheme to create a correspondence between the bit-representation of values (denoted q below, for "quantized value") and their interpretation as mathematical real number (denoted r below, for "real value") [43]. This scheme only uses integer-only arithmetic during inference and not fixed-point arithmetic which is often incorrectly stated in other sources. The mapping of integers q to the real number r is in the form:

$$r = S(q - Z) \tag{5.1}$$

Equation 5.1 represents the quantization scheme and the constants S and Z the quantization parameters. Integer q is quantized as an 8-bit integer. Scaling the quantized value to a real value is performed by the arbitrary positive real number S. The scale is calculated using Equation 5.2, where $r_{max} \geq 0$ and $r_{min} < 0$ represent the maximum and minimum real value respectively.

$$S = \frac{|r_{max}| + |r_{min}|}{255} \tag{5.2}$$

Constant Z (for "zero-point") equals the quantized value for the real value r = 0 and is of the same type as q. The zero-point is calculated using Equation 5.3. Subtracting Z from q allows the real value r = 0 to be representable by a quantized value.

$$Z = \left\lceil \frac{|r_{min}|}{S} \right\rceil - 128 \tag{5.3}$$

To illustrate how the scheme works, an example is shown in Figure 5.2. Real values r with r_{min} of -10 and r_{max} of 90 are represented by the top line. On the bottom, the corresponding 8-bit representation q is shown where r_{min} corresponds with -128 and r_{max} with 127. Computing the scale gives $S = 100/255 \approx 0.39$, and zero-point $Z = 26 - 128 = -102$. Real values outside range (r_{min}, r_{max}) are clamped.

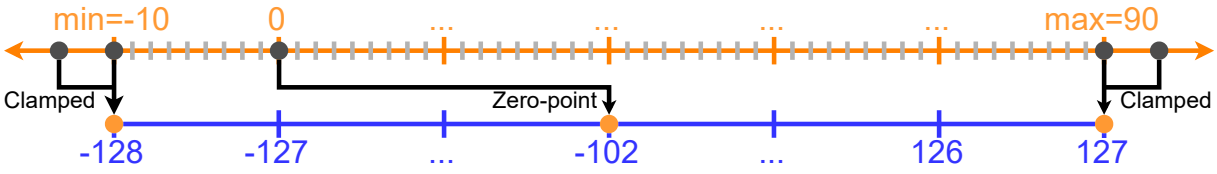


Figure 5.2: TensorFlow Lite quantization scheme example with $min = -10$, $max = 90$, $Z = -102$, and $S \approx 0.39$.

5.3.2 Algorithm

Designing an accelerator that is compatible with TFLM requires a deep understanding of how the quantization scheme is embedded into the convolution algorithm. This section explains this by reducing the algorithm to a simple convolution between a single filter $W[R][S]$ and its corresponding input fmap $I[H][W]$ creating an output fmap $O[M][E][F]$, see Equation 5.4. Besides bias $B[M]$, different channels and filters are omitted. Stride is represented by U.

$$O[M][E][F] = B[M] + \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} I[U \cdot E + i][U \cdot F + j] \cdot W[i][j] \tag{5.4}$$

By representing the real output value and the quantized value as r_3 and q_3 respectively, the output can be represented as:

$$r_3 = S_3(q_3 - Z_3) \tag{5.5}$$

Applying the same representation to input fmap I and filter W as q_1 and q_2 respectively:

$$S_3(q_3[M][E][F] - Z_3) = B[M] + \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} S_1(q_1[U \cdot E + i][U \cdot F + j] - Z_1) \cdot S_2(q_2[i][j] - Z_2) \tag{5.6}$$

This equation can be rewritten as:

$$q_3[M][E][F] = Z_3 + P(B[M] + \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} (q_1[U \cdot E + i][U \cdot F + j] - Z_1) \cdot (q_2[i][j] - Z_2)) \tag{5.7a}$$

$$P = \frac{S_1 S_2}{S_3} \tag{5.7b}$$

Multiplier P is defined in Equation 5.7b and is the only non-integer in the equation. Since S_1, S_2, and S_3 are constants, P can be computed offline. P is always between the interval (0,1) [43] and can therefore be expressed in the normalized form:

$$P = 2^{-n} M_0 \tag{5.8}$$

The normalized multiplier M_0 is expressed as a fixed-point multiplier in the interval $[0.5, 1)$ and n as a non-negative integer. Meanwhile, 2^{-n} can be implemented with a bit-shift. Note that weights W represented as $q_2 - Z_2$ in Equation 5.7a are constant and therefore can also be computed offline reducing the number of online computations even more.

The complete algorithm is given in Algorithm 2.

Algorithm 2: TensorFlow Lite convolution naive implementation.

Input : I, W, B, $Z_{in}, Z_{out}, M_0, n, max_{out}, min_{out}, pad_h, pad_w, U_h, U_w$
$I[N][C][H][W]$: Input Fmaps
$W[M][C][R][S]$ Filter Weights
$B[M]$: Biases
Z_{in}: Input Zero-point
Z_{out}: Output Zero-point
$M_0[M]$: Normalized Multiplier
$n[M]$: Normalized Multiplier Shift
max_{out}: Maximum Output Range
min_{out}: Minimum Output Range
pad_h: Padding Height
pad_w: Padding Width
U_h: Stride Height
U_w: Stride Width

Output: O
$O[N][M][E][F]$: Output Fmaps

begin
  for n *in* N **do**
    for e *in* E **do**
      $in_{y0} = (e \cdot U_h) - pad_h$
      for f *in* F **do**
        $in_{x0} = (f \cdot U_w) - pad_w$
        for m *in* M **do**
          $acc = 0$
          for r *in* R **do**
            $in_y = in_{y0} \cdot r$
            for s *in* S **do**
              $in_x = in_{x0} \cdot s$
              for c *in* C **do**
                if $(in_x \geq 0)$&&$(in_x < H)$&&$(in_y \geq 0)$&&$(in_y < W)$ **then**
                  $acc = acc + (I[n][c][in_y][in_x] + Z_{in}) + W[m][c][r][s]$
                end
              end
            end
          end
          $acc = acc + B[m]$
          $acc = 2^{-n[M]} \cdot M_0[M] \cdot acc + Z_{out}$
          $acc = \text{min}(\text{max}(acc, max_{out}), min_{out})$
          $O[n][m][e][f] = acc$
        end
      end
    end
  end
end

6 FPGA ACCELERATOR DESIGN AND IMPLEMENTATION

In the previous chapter it was identified that computing convolutional layers causes a bottleneck in DNN inference. Analyzing the convolutional algorithm shows that most of the computations involve MAC operations. Each MAC operation produces one partial sum (psum) by multiplying an input (ifmap) with a weight (filter) and accumulating it with the previous psum. This generates a significant amount of data movement.

This chapter focuses on accelerating the algorithm by designing and implementing an FPGA-based hardware accelerator. The accelerator improves performance by parallelizing MAC operations so that computational power is increased. Increasing computational power, however, can introduce bandwidth limitations limiting MAC utilization. To solve this, an efficient schedule of operations is created such that data reuse is exploited. Exploiting data reuse is increased by introducing a memory-level hierarchy between the MACs and DRAM.

The level of MAC utilization and data reuse depends on the size and shape of a DNN layer. Opportunities for data reuse are determined by the size of the filter, number of channels, number of filters, etc [44]. Therefore, the schedule of operations (mapping) changes across different DNN layers. Finding the most optimal mapping for each layer is accomplished by the Timeloop mapper tool, as described in Section 6.2. It tries to find the best mapping that optimizes MAC utilization and data reuse.

The accelerator is based on an existing accelerator called Eyeriss [1][2]. Eyeriss is a state-of-the-art accelerator for deep convolutional neural networks. The hardware architecture is based on the Row Stationary (RS) dataflow.

This chapter will first, in Section 6.1, provide a detailed overview of the RS dataflow. Second, in Section 6.2, the Timeloop mapper tool is explained. Then, in Section 6.3, the Eyeriss architecture design is discussed. After designing the architecture, it is implemented in Section 6.4 using High-Level Synthesis compatible C++. A custom-made configuration tool configures the accelerator and is discussed in Section 6.5. Now that the accelerator is implemented, it needs to be integrated within the complete systems which is discussed in Section 6.6. Section 6.7 then describes how the accelerator implemented in C++ is synthesized using Catapult. Finally, in Section 6.8, the workflow of integrating the Catapult generated accelerator in RTL into the complete system is presented.

6.1 Row Stationary Dataflow

Accelerator dataflows try to exploit data reuse in the architecture. This architecture consists of an array of Processing Elements (PEs), with a MAC for computation and a register file (RF) for storage. An additional memory level, called the Global Buffer (GLB), is added to create shared local storage for the PE array. Many techniques exist that exploit data reuse, these can be categorized into the following dataflows [9]: Weight Stationary (WS), Output Stationary (OS), Input Stationary (IS), and Row Stationary (RS).

The WS dataflow is designed to minimize the fetching of weights by keeping them stationary in the RF of each PE. Each PE reuses the weight(s) stored in the RF, thereby maximizing the filter reuse in the RF. While this maximizes filter reuse, ifmaps and psums still need to be loaded each time and the produced psum need to be sent back to memory.

An accelerator designed with the OS dataflow keeps psums stationary by accumulating them locally in the RF. The data reuse of all other data types is not optimized. The same applies to the IS dataflow, except that the input is kept stable instead of psums.

All of the previously explained dataflows optimize for only one type of data. The RS dataflow, however, optimizes the data reuse with respect to all data types. This reduces energy usage [1] and bandwidth requirements. The next section explains in detail how this dataflow functions.

6.1.1 Approach

RS uses a systematic approach to optimize for data types simultaneously. It first reduces the high-dimensional convolution to multiple 1-D convolutions that can run in parallel. By stacking multiple 1-D convolutions, computing a 2-D convolution is realized. Finally, there are multiple techniques to compute convolutional problems with dimensions beyond 2-D. These three steps are further explained below.

1-D Convolution in a PE

1-D convolutions are mapped to individual PEs, where each PE operates on one row of filter weights and one row of ifmaps. It then generates one row of psums keeping the *row pair stationary* in a PE. A PE stores the data in Scratch Pads (SPads), which are blocks of memory with some control logic built into them. The sizes of these SPads depend on the filter row size (S), but not the ifmap row size (W) since only a sliding window of data is retained at a time. Only one psum is stored to make local accumulation possible. The size of these SPads can be increased to realize computations beyond 2-D, but this is elaborated later on.

Figure 6.1 shows an example of a 1-D convolution in a PE. All data types are shifted into the PE in a windowed fashion indicated by the black boxes around the values. The psums shifted in can be psums generated by another PE or a bias value. Step 1 computes the psum of the convolution by shifting values out of the SPads and storing them back in front of the queue for both filters and ifmaps. A computed psum is shifted back into the psum SPad. After repeating this process for the total window length, a new ifmap value is shifted in and the process repeats itself, i.e., steps 2 and 3.

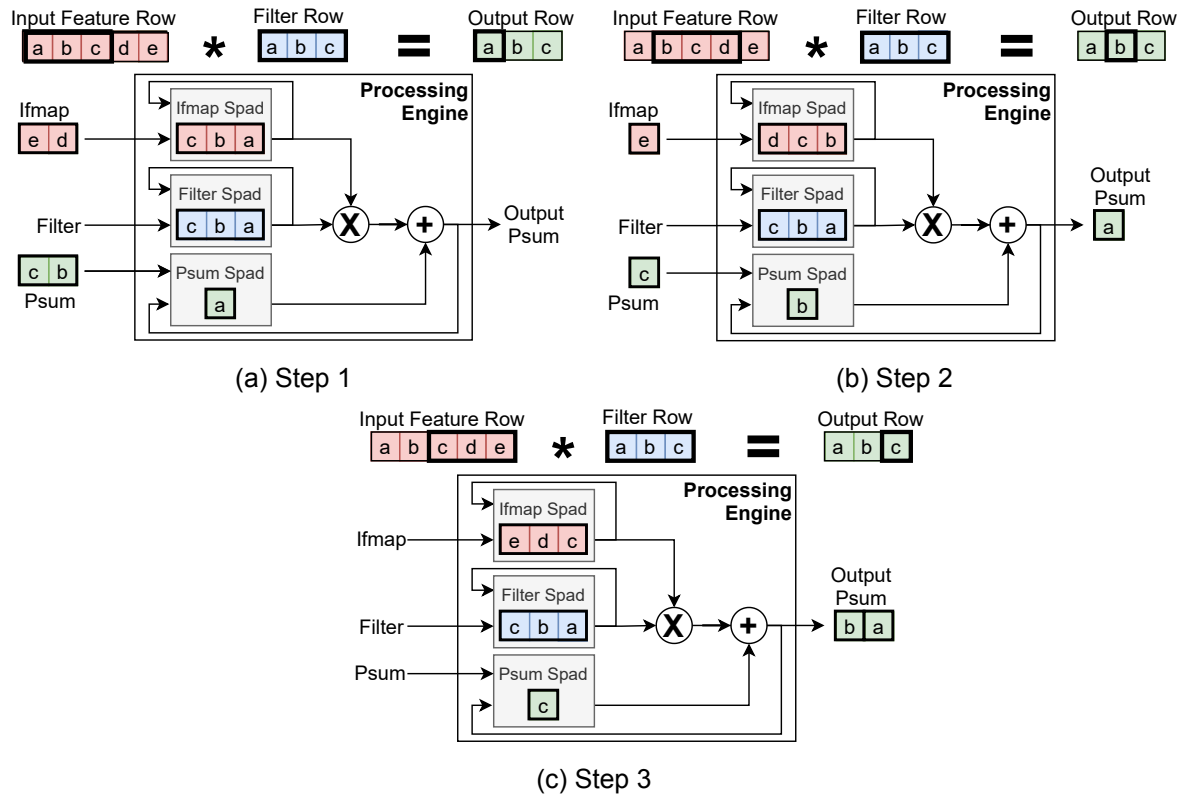


Figure 6.1: 1-D convolution in a PE.

2-D Convolution PE Set

A 2-D convolution is composed of multiple 1-D convolutions. So by combining multiple 1-D convolutions, i.e., multiple PEs, a PE set is created that can perform 2-D convolutions as illustrated in Figure 6.2. Calculating a row of psums is done by vertically stacking PEs which accumulate their results together, indicated by the red arrows. Copying this row of vertical stacked PEs horizontally allows multiple output rows to be calculated simultaneously.

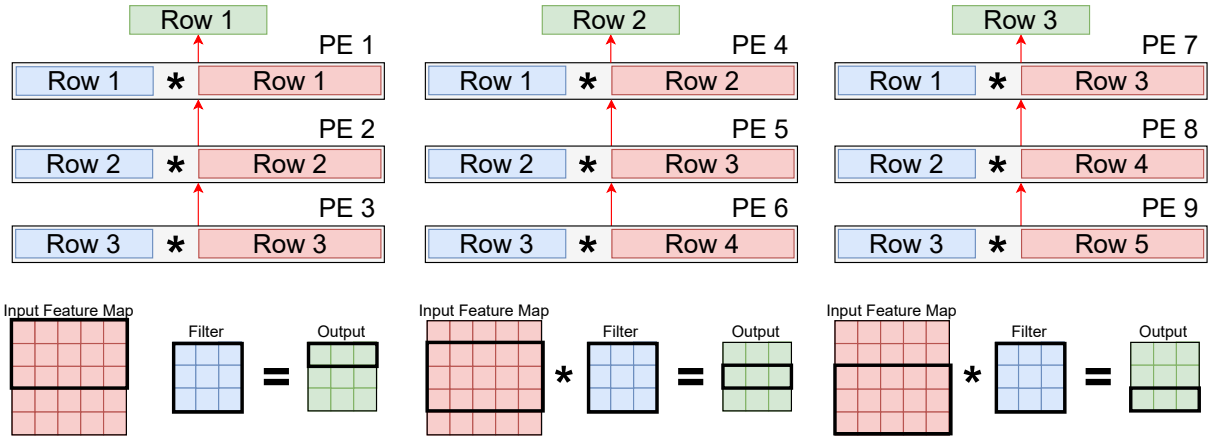


Figure 6.2: 2-D Convolution in a PE Set.

Data reuse in a PE set is different for each data type. The filter row values are shared across multiple PEs horizontally. Rows of ifmaps are reused across PEs diagonally and psums across the vertical axis. The PE set dimensions determine the amount of data reuse within the set.

Dimensions Beyond 2-D in PE Array

Three additional dimensions have an impact on the computation. These dimensions include the batch size (N), the number of channels (C), and the number of filters (M). The batch size is set to one since the DNN only computes predictions based on a single input image, leaving only two additional dimensions.

The first technique to compute beyond 2-D is to fit multiple PE sets in a PE array. Each set runs on r different channels and t different filters, resulting in the array being able to fit r x t sets that run in parallel. Data reuse is further increased since ifmaps are shared every t sets and psums are accumulated over every r sets. Figure 6.3 provides an example, with M=C=R=t=r=2 and E=3, where each set is colored differently. Psum are accumulated over r, meaning that block one (blue and yellow) and block two (green and red) accumulate psums.

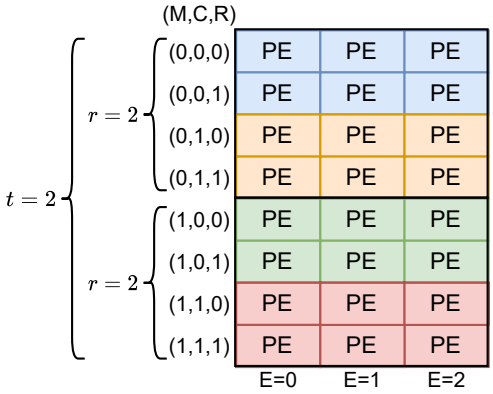


Figure 6.3: Row-Stationary dataflow mapping multiple PE Sets in the PE Array.

Another technique that can be exploited is to run multiple 2-D convolutions sequentially in a PE set. Increasing the SPad sizes allows for two additional data reuse opportunities. First, a PE may run on p different filters by increasing the psum and filter SPad sizes with the result that the same ifmap can be used for multiple filters. Second, storing q channels of filters and ifmaps in a PE allows for sequentially accumulating on the same psum over different channels. This requires an increase of the filter and ifmap SPad. The total capacity of each SPad is calculated as follows: filter SPad size: p x q x S, ifmap SPad size: q x S, and psum SPad size: p.

6.1.2 Dataflow

The RS dataflow as implemented in this project is given in Figure 6.4. Two types of loops describe how data is stored and sent to the individual PEs. Storing data over dimensions is represented by for-loops. The distribution of data over PEs is represented by parallel-for loops. Three levels of memory hierarchy, DRAM, the Global Buffer, and SPads, exist to store data. Dimensions are split over these levels, for example, dimension E is split in two over E4 and E2. The mapper assigns specific values to the loop bounds marked red.

```
Input Fmaps:      I[N][C][H][W]
Filter Weights:   W[M][C][R][S]
Output Fmaps:     O[N][M][E][F]

//DRAM: storage loops
for (e4=0; e4<E4; e4++) {
  for (n4=0; n4<N4; n4++) {
    for (m4=0; m4<M4; m4++){
      for (c4=0; c4<C4; c4++){
        //Global Buffer: storage loops
        for (m3=0; m3<M3; m3++){
          for (n3=0; n3<N3; n3++) {
            for (c3=0; c3<C3; c3++){
              for (f3=0; f3<F3; f3++){
                //NoC: spatial loops (X-axis)
                parallel-for (e2=0; e2<E2; e2++){
                  //NoC: spatial loops (Y-axis)
                  parallel-for (m1=0; m1<M1; m1++) {
                    parallel-for (c1=0; c1<C1; c1++){
                      parallel-for (r1=0; r1<R1; r1++) {
                        //SPad: storage loops
                        for (s0=0; s0<S; s0++){
                          for (c0=0; c0<C0; c0++) {
                            for (m0=0; m0<M0; m0++){
                              O[n4*N3+n3][m3*M2*M1*M0+m2*M1*M0+m1*M0+m0][e4*E2+e2][f0] +=
                                W[m3*M2*M1*M0+m2*M1*M0+m1*M0+m0][c4*C3*C1*C0+c3*C1*C0+c1*C0+c0][r1][s0] *
                                I[n4*N3+n3][c4*C3*C1*C0+c3*C1*C0+c1*C0+c0][h][w];

                              //h = (e4*e2)*Wstride + r1*Wdilation
                              //W = f0*Hstride + s0*Hdilation
}}}}}}}}}}}}}}}
```

Figure 6.4: Row Stationary dataflow definition.

6.2 Timeloop

The hardware of DNN accelerators allows operations to be partitioned and scheduled for computation, and how data is stored in different memory hierarchies. These properties are described in a dataflow, see Figure 6.4 for the dataflow used in this project. Finding the most optimal way to schedule operations and stage data on the architecture for computing a DNN layer, called a mapping, is achieved by Timeloop [45]. Timeloop constructs a space of valid mappings, i.e., the mapspace, and uses a user-defined optimization goal to find the most optimal mapping. A mapping can be optimized for performance, i.e., number of cycles, energy efficiency, or both.

The tool flow, see Figure 6.5, requires the user to describe a workload as a DNN layer specification, the accelerator architecture, constraints implied by the architecture and constraints on the possible mappings strategies, and finally an optimization goal.

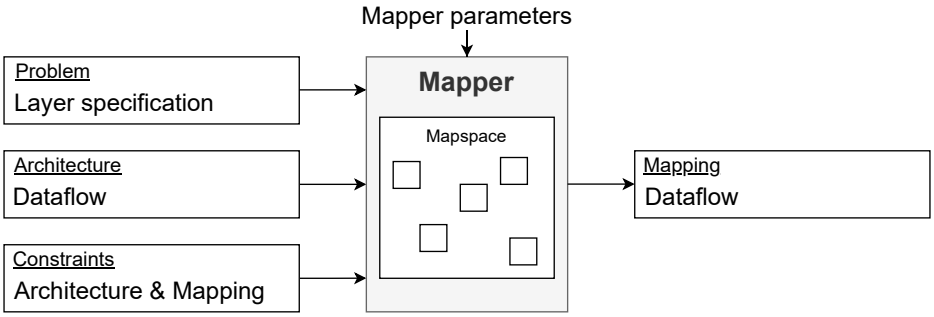


Figure 6.5: Timeloop tool flow.

6.2.1 Workload

Timeloop analyzes and maps one DNN layer at a time which requires Timeloop to be run sequentially on each layer to evaluate a complete network. The workload is specified by a dataflow such as shown in Figure 6.6a. How the output is computed, depends on the loop bounds indicated by the red letters and array indexing. Users need to specify these constructs for Timeloop to correctly map the workload. A small part of the workload description is shown in Figure 6.6b which describes loop bounds and Figure 6.6c describing indexing of the weight array.

```
for n = [0:N):
 for m = [0:M):
  for f = [0:F):
   for e = [0:E):
    for s = [0:S):
     for r = [0:R):
      for c = [0:C):
       Output[n][m][e][f] +=
         Weight[m][c][r][s] *
         Input[n][c][h][w]
```

(a) Workload dataflow

```
instance:
  C: 3
  M: 32
  N: 1
  F: 208
  E: 208
  R: 3
  S: 3
```

(b) Loop bounds

```
data-spaces:
- name: Weights
  projection:
    - [ [M] ]
    - [ [C] ]
    - [ [R] ]
    - [ [S] ]
```

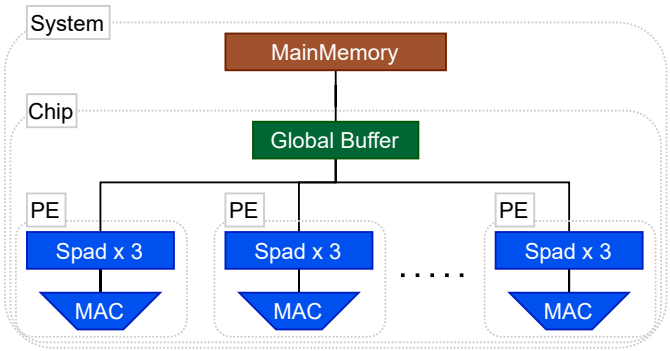
(c) Array indexing

Figure 6.6: Timeloop workload definition example.

6.2.2 Architecture

An accelerator architecture is modeled in Timeloop by specifying the hardware organization as the topology of interconnected compute and storage units [45]. The model is a hierarchical tree of storage elements and arithmetic units at the leaves. Each storage element specifies architectural specifications such as memory depth, memory width, bandwidth, etc.

Figure 6.7 depicts the Eyeriss architecture and its corresponding architecture definition. As can be seen in Figure 6.7a, the architecture uses a three-level memory hierarchy with one main memory modeled as DRAM, a Global Buffer, and in each Processing Element (PE) three scratch pads. Figure 6.7b illustrates how the Eyeriss architecture is modeled with a 3x4 PE array.



(a) Eyeriss basic architecture

```
architecture(
  subtree(
    - name: DRAM
      - width: 32
      - word-bits: 8
    subtree(
      - name: Global Buffer
        - sizeKB: 128
        - width: 32
        - word-bits: 8
      subtree(
        - name: PE[0..11]
          - local(
            - name: ifmap_spad
              - sizeB: 24
              - width: 9
              - meshX: 4
            - name: weights_spad
              - sizeB: 288
              - width: 8
              - meshX: 4
            - name: psum_spad
              - sizeB: 32
              - width: 32
              - meshX: 4
            - name: MAC
              - width: 32
              - meshX: 4
          ) //local
        ) //subtree PE
      ) //subtree Global Buffer
    ) //subtree DRAM
```

(b) Definition

Figure 6.7: Timeloop Eyeriss architecture with a 3x4 Processing Element array.

Each memory hierarchy introduces a subtree and under the final subtree, an array of 12 PEs is defined. The model of each PE is embodied within the local enclosing. Within the local enclosing, three additional memory elements are defined, and finally a MAC unit for the actual calculation. The meshX parameter defines the width of the array, this allows Timeloop to identify that the PE array should have a dimension of 3x4.

6.2.3 Constraints

Timeloop, by default, has the complete flexibility to partition and schedule arithmetic operations and data movement across the hardware resources. However, architecture constraints limit this flexibility, and users can introduce additional constraints via mapspace constraints. Three types of constraints exist which are: temporal, spatial, and bypass constraints. Next to these constraints, there are factors that fix values for loop bounds and permutations that specify loop ordering within a storage level.

Temporal constraints affect the data access patterns at a storage level. In Figure 6.8b, no data is stored in the Global Buffer along dimensions S, R, E, and N, Timeloop can store data over all other dimensions. Spatial constraints limit the partitioning of the workload across the spatial dimension. The Global Buffer for example as shown in Figure 6.8a, only allows spatial partitioning in the E and M dimension. Finally, bypass dictates whether a data type is stored or bypassed. The architecture constraints in Figure 6.8a only allow the weights scratch pad to store weights. spatial

```
architecture_constraints:
  targets(
    - target: weights_spad
      - type: bypass
      - bypass: [Inputs, Outputs]
    - target: Global Buffer
      - type: spatial
      - permutation: EMCFNSR
      - factors: C=1 F=1 N=1
                 S=1 R=1
  ) //targets
```

(a) Architecture constraints

```
mapspace_constraints:
  targets(
    - target: DRAM
      - type: temporal
      - permutation: CMENSRF
      - factors: N=1 S=1 R=1 F=1
    - target: Global Buffer
      - type: temporal
      - permutation: CFMNSRE
      - factors: S=1 R=1 E=1 N=1
  ) //targets
```

(b) Mapping constraints

Figure 6.8: Timeloop architecture and mapping constraints.

6.2.4 Mapping

After running Timeloop with all the user-provided files, a mapping is created in the form of a .txt file. Figure 6.9 illustrates what such a mapping looks like. For each level, the storage requirements are provided, for example, there are 72 weights stored in the weights scratch pad. Furthermore, loop bounds for both temporal and spatial storage are created. The axis on which a dimension is unrolled is indicated after the loop by either Spatial-X for unrolling along the x-axis or Spatial-Y for the y-axis.

```
DRAM [ Weights:864 Inputs:50700 Outputs:524288 ]
-------------------------------------------------
| for E in [0:8)

Global Buffer[ Inputs:7020 Outputs:65536 ]
-----------------------------------------
|   for F in [0:128)
|     for E in [0:16) (Spatial-X)
|       for M in [0:4) (Spatial-Y)
|         for R in [0:3) (Spatial-Y)

ifmap_spad [ Inputs:9 ]
-----------------------
|           for R in [0:1)

weights_spad [ Weights:72 ]
---------------------------
|             for S in [0:3)
|               for C in [0:3)

psum_spad [ Outputs:8 ]
-----------------------
|               for M in [0:8)
```

Figure 6.9: Timeloop created mapping.

6.3 Architecture Design

The Eyeriss accelerator is taken as the basis of the architecture design. Parts of its architecture are described in two research papers [1][2] and the PhD thesis [44] it originates from. This information together with knowledge obtained in Section 5.3.2 about the CONV_2D kernel and other sources resulted in the design presented in this section.

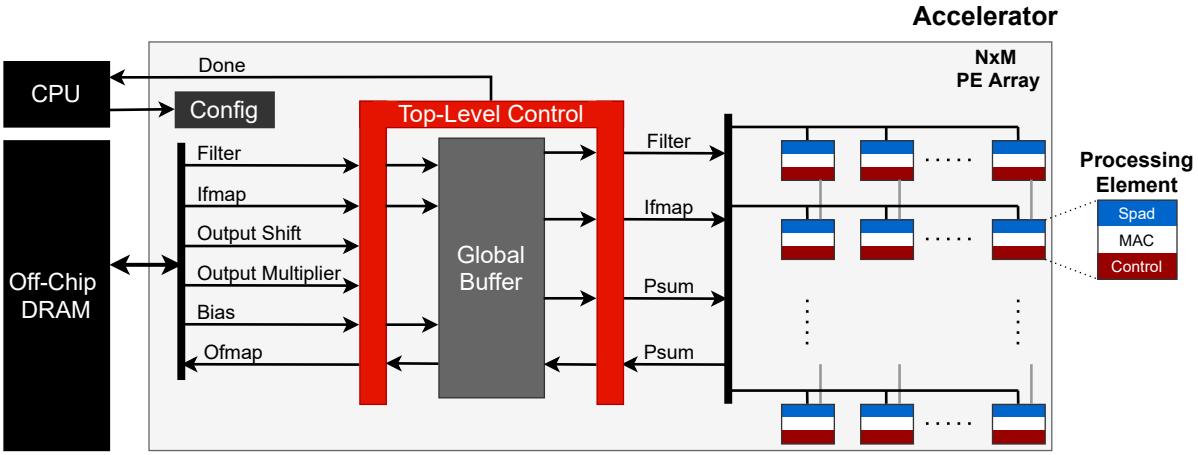


Figure 6.10: Architecture overview.

Figure 6.10 presents an overview of the architecture. Off-chip DRAM stores all necessary data consisting of filters, ifmaps, biases, and for quantization the output shift and multiplier. The GLB stores filters, ifmaps, and biases which are updated after the PE array finished its local SPad loop (Figure 6.4). After computing a psum, it is sent back to DRAM over the *Ofmap* interface. The CPU can load an array containing configuration data with information on how to compute a certain layer to the accelerator and thereby triggering the accelerator to start. When the accelerator has finished computing all ofmaps of a certain layer, the CPU is interrupted via the *Done* signal.

The Top-Level Control is responsible for keeping the utilization of the PE array as high as possible by fetching data from DRAM and storing it in the GLB. It then pushes available data from

the GLB in the PE array. Since all data passes through the Top-Level Control, it is most efficient in terms of area to use this central block for applying the quantization scheme. Incoming filters are directly quantized before storing them in the GLB. Outgoing psums are quantized when they leave the GLB on their way back to DRAM.

6.3.1 Network-on-Chip

The PE array achieves high data reuse because of the Network-on-Chip (NoC) that manages data delivery between the GLB and the individual PEs. It supports the spatial data delivery patterns defined by the RS dataflow. These patterns allow for computations beyond 2D in the PE array but it also takes care of stride (U). Stride results in ifmap values delivery skipping certain rows in the array.

The NoC differs from traditional NoC architectures where multiple segments of routers decide on whether to forward a received packet horizontally, vertically, or to the local PE [46]. This architecture implements a simpler NoC, which is comprised of three types of networks. These networks are further elaborated on in the next sections.

Global Input Network

The Global Input Network (GIN) allows the GLB to sent ifmaps, filters, and psums to the PE array. All data types have their own GIN enabling separate data delivery for each type. Figure 6.11 shows how the GIN architecture for a single data type operates. The network consists of two buses, the global X-bus connecting all PEs on a row together, and a global Y-bus that links all X-buses together. Data sent from the GLB is augmented with a row and column id. Each global X-bus has its own row id controlled by a Multicast Controller (MC) right after data branches from the Y-bus. Before the MC can pass received data, it checks if the row id matches a configurable id. Data is dropped if this is not the case. Data delivery to individual PEs is managed by MCs between the X-bus and a PE. These MCs compare the column id with a pre-configured id before data is passed to the PE.

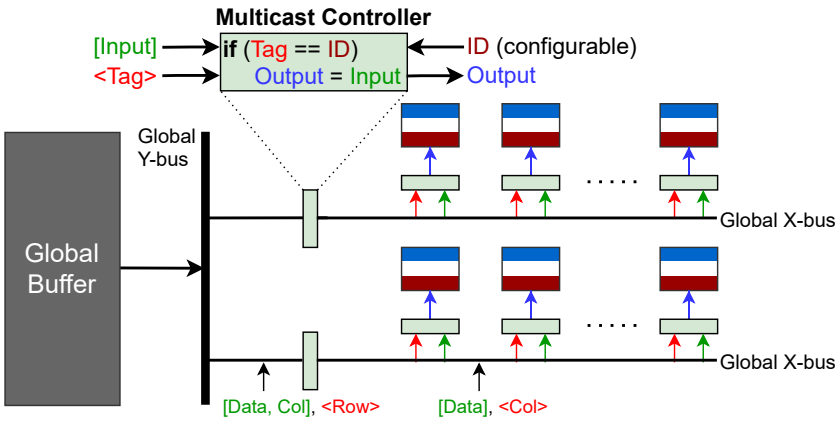


Figure 6.11: Global Input Network architecture [1]

The MCs enable data to be delivered to individual PEs (unicast), a group of PEs (multicast), or all PEs (broadcast).

Configuring the row and column ids of the MCs is done by the Configurator tool that is described in Section 6.5. This tool parses the mapping created by Timeloop and creates a char array containing the ids. Id configuration depends on the data type and layer specifications. For

ifmaps, the row id equals $c1$ since this is the only dimension mapped on the Y-axis influencing its vertical mapping. The column ids allow for diagonal mapping and are calculated as follows:

$$column\ id = e2 \cdot U + r1$$

An example of this principle is shown in Figure 6.12a. This example corresponds with the example presented in Figure 6.3 except for dimension E, which is set to 6. The red boxes indicate that a single data value is multicasted to multiple PEs sharing the same row and column id.

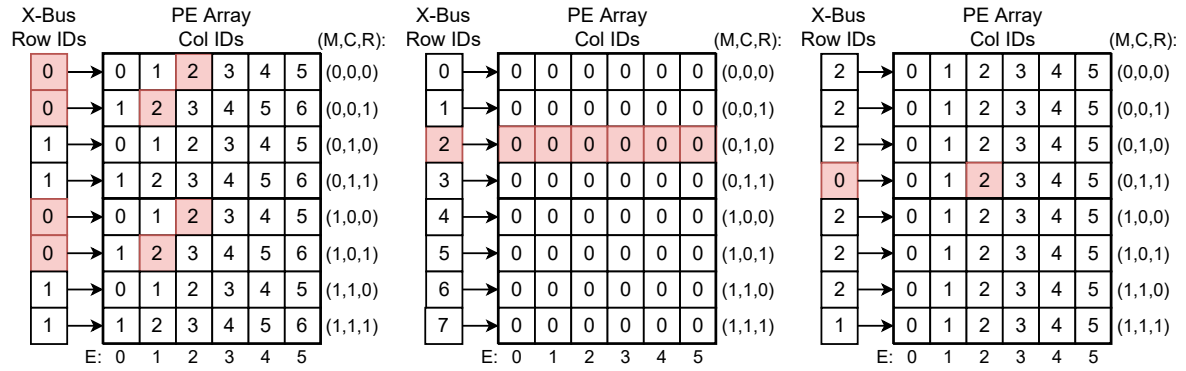


Figure 6.12: PE array configured with row and column ids indicating delivery patterns of data.

Data reuse for filter data is illustrated in Figure 6.12b. Since filter data is reused horizontally, the column id is set to 0. Each row has its own id:

$$row\ id = m1 \cdot C1 \cdot R + c1 \cdot R + r1$$

Psum id configuration differs from the other data types because each value is sent only to a single PE, i.e., unicast, see Figure 6.12c. Every r sets accumulate psums, therefore, only the first PE, i.e., the first PE in the accumulation chain, needs to receive a psum from the GLB. All PEs in a row get a different column id which is set to $e2$ for the column a PE is located in. Sending a psum only to the first PE in a set requires the row ids to be set to:

$$row\ id = \begin{cases} m1, & last\ row\ of\ a\ PE\ Set \\ M, & otherwise \end{cases}$$

Global Output Network

The Global Output Network (GON) collects the psums generated by the PE array. The GON architecture is originally designed as a GIN but with a reverse transfer of data. Due to limited time, the GON is implemented differently by grouping all psums in a 2D C array. The individual psums can then be easily accessed by the Top-Level Control block.

Local Network

Between two PEs that are on the same column with consecutive rows, an interface, called the Local Network (LN), is implemented that allows psums to flow from the bottom PE to the top PE directly. Configuration determines if a PE receives its psum from the GIN or the LN. In addition to the input being configurable, the output is also configurable. This is because a PE can send the produced psum to either the GON or the LN. Looking back at Figure 6.12c, it becomes clear that the only PEs using the GIN as input are the PEs on the first row within a PE set. All other

PEs are configured to take their input from the LN. The opposite is true for the output. Hence, only the PEs in the last PE row of a PE set send the data to the GON. All other PEs in the set are configured to send data to the LN.

6.4 Architecture Implementation

The accelerator is implemented in High-Level Synthesis compatible C++ which is synthesized to RTL using Catapult. The design is divided into four blocks that run concurrently with one another in a hierarchical design. These blocks, represented by the Config, Top-Level Control, Global Buffer, and the PE array block, all implement their own hierarchy, except for the Config block, with sub-block running in parallel. This approach allows for higher throughput by allowing blocks to run in parallel and a reduction in development time since each block can be tested separately. Another reason for hierarchy according to the Catapult manual is that it reduces synthesis runtime. Catapult performs best on blocks that are between about 10K and 200K gates.

6.4.1 Config

The Config block continually inspects the AXI4 slave memory that the CPU uses to load configuration data. The CPU indicates that the accelerator should start by setting the last bit in the last configuration memory location. After the Config block detected that this bit is set, it fetches all other available memory from the AXI4 slave block. It then extracts parameters from the data and configures corresponding parameters. An example showing the column MCs getting configured is shown below:

```
mc_config<colfType, colType> config_col[HEIGHT][WIDTH];
config_col[0][0].fmap_id = config[1].template slc<4>(12);
config_col[0][0].filter_id = config[1].template slc<1>(11);
config_col[0][0].psum_id = config[1].template slc<3>(8);
//Parameterisation of other indexes omitted
#pragma hls_unroll yes
CONFIG_COL_HEIGHT: for (int h=0; h<HEIGHT; h++){
#pragma hls_unroll yes
  CONFIG_COL_WIDTH: for (int w=0; w<WIDTH; w++){
    config_col_out[h][w].write(config_col[h]);
}}
```

This example shows, for example, that the column MC at location (0,0) is configured using the data stored in config array index one at lsb bit 12 with a width of 4 bits. When all parameters are decoded from the config array, they are written to the individual column MCs using a separate channel.

6.4.2 Top-Level Control and Global Buffer

The Top-Level Control and GLB are tightly coupled, therefore they are explained together in this section. Figure 6.13 presents the implementation overview of the two blocks with each block having sub-blocks implemented in them.

Data stored in DRAM is fetched by Fill Address Generators (AGENs), inspired from [47], that generate addresses based on pre-configured parameters. Generated addresses are different for each data type, therefore, all data types have their own Fill AGEN. After data is retrieved from DRAM, it is stored in the GLB. Since ifmaps and filters are simply pushed into the PE array, storing them in a circular buffer suffices. Psums on the other hand, are first fetched

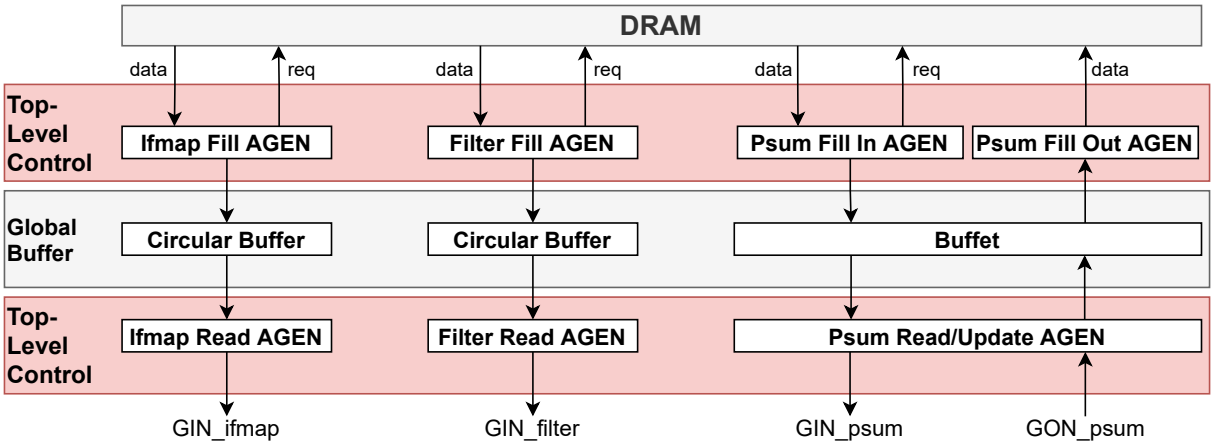


Figure 6.13: Top-Level Control and Global Buffer implementation overview.

from DRAM as a bias, then get pushed in the PE array, after computation, get retrieved from the array and update the GLB, finally when all computation on a psum has been completed, it is sent back to DRAM. Storage for psums should therefore support filling, reading, updating, and shrinking of data. Support for this type of storage is realized using a Buffet, invented in [47].

When data is available in the GLB, it gets pushed to Read AGENs. These AGENs know, based on pre-configuration and an internal state-machine, how to label data with a row and column id. Both ifmap, filter, and psum Read AGENs implemented a similar architecture. But the psum Read AGEN is modified to support retrieving psums for the PE array and sending them back to the Buffet.

Fill AGEN

Fill AGENs do not generate addresses at startup, but only start when indicated via the configuration signal. After receiving this signal, it enters a structure similar to the for loop structure of the RS dataflow which is used to correctly generate addresses for fetching data from DRAM. The following code for the Filter AGEN provides a better understanding of this structure is implemented:

```
//DRAM: storage loops
for (dType_M e4=0; e4<_config.df.E4; e4++){
  for (dType_M m4=0; m4<_config.df.M4; m4++){
    //Global Buffer: storage loops
    for (dType_M m3=0; m3<_config.df.M3; m3++){
      for (dType_C c3=0; c3<_config.df.C3; c3++){
        //SPad: storage loops
        for (dType_S s0=0; s0<_config.df.S; s0++){
          for (dType_C c0=0; c0<_config.df.C0; c0++){
            for (dType_M m0=0; m0<_config.df.M0; m0++){
              //NoC: spatial loops (Y-axis)
              for (dType_M m1=0; m1<_config.df.M1; m1++){
                for (dType_C c1=0; c1<_config.df.C1; c1++){
                  for (dType_R r1=0; r1<_config.df.R; r1++) {
                    dType_M m_index = m0 +
                                      m1 * _config.df.M0 +
                                      m3 * _config.M1_M0 +
                                      m4 * _config.M3_M1_M0;

                    dType_C c_index = c0 +
```

```
                                c1 * _config.df.C0 +
                                c3 * _config.C1_C0;

                    dType data = data_in[m_index][r1][s0][c_index];
                    data_out.write(data);
}}}}}}}}}}
```

The structure loops over all dimensions influencing a particular datatype, in this case, the dimensions affecting filters. Received data is written to the GLB via an interconnecting channel. Index m and c are computed within the inner loop using pre-computed values, for example, _config.M3_M1_M0 represents the pre-computed value $M3 \cdot M1 \cdot M0$. Notice that the SPad and NoC loop have swapped place with each other compared to the original RS dataflow definition. This allows for higher throughput since the data is first sent spatially in contrast to first filling a single PE when the SPad storage loops were to be placed as inner loops.

Fill Out AGEN

The Psum Fill Out AGEN architecture is the same as implemented for the Fill AGENs but receives data from the GLB and sends it back to DRAM with the corresponding memory address.

Circular Buffer

The circular buffer operates like you would expect with a read (head) and write (tail) pointer. The code below shows the run function of the circular buffer hierarchical class. Data originating from a Fill AGEN enter the block via the data_in channel and read commands from a Read AGEN via the read channel. The size of both channels is read to determine if data is present. Writing or reading data depends on the buffer status and if data is present.

```
#pragma hls_design interface
void CCS_BLOCK(run)(ac_channel<dType> &data_in,
                    ac_channel<bool> &read,
                    ac_channel<dType> &data_out){
  bool data_available = data_in.size()>0;
  bool read_available = read.size()>0;

  //Write data to buffer
  if (!_full && data_available)//Logic omitted
  //Read data from buffer
  if (!_empty && read_available)//Logic omitted
  //Logic omitted
}
```

Buffet

The Buffet storage element has support for filling, reading, updating, and shrinking. It functions as a circular buffer with a read and write pointer but introduces two additional pointers. An update pointer keeps track of the next index to be updated. Reading a value results in the read pointer being incremented but the first read index is stored by the read_base pointer, which is only incremented after shrinking, this enables for reading after updating a value. The read pointer resets itself to the read_base pointer after all psums, i.e., $M1 \cdot M0 \cdot E2$ psums, are sent to the PE array. Figure 6.14 shows a buffet operation example.

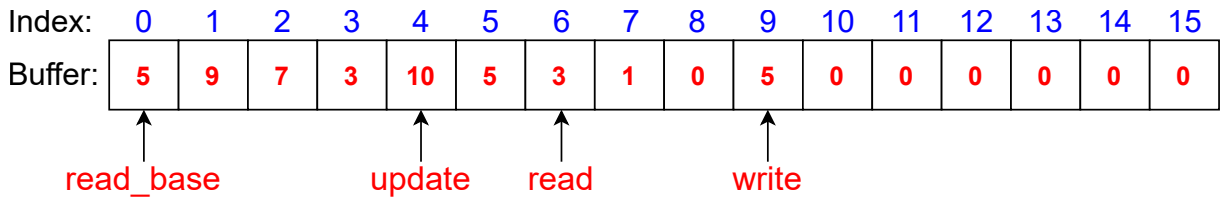


Figure 6.14: Buffet operation example.

The lifetime of a piece of data is described with the following regular expression:

$$Fill \rightarrow (Read \rightarrow Update?)^* \rightarrow Shrink$$

Shrinking involves sending n amount of values out of the buffet to the Fill Out AGEN, where n equals $M1 \cdot M0 \cdot E2$. If the Buffet should read, update, or shrink is determined by an external block, in this case, the Psum Read/Update AGEN block, via command channels. The Buffet handles read and update commands but the external block is responsible for correctly using the shrink command, i.e., shrinking only filled indexes.

Read AGEN

Read AGENs receive data from the GLB and augment it with a row and column id. Ids are determined based on pre-configuration and a state machine. The basic structure of an AGEN is shown below which illustrates this through code from the Filter Read AGEN:

```
if (data_in.size()>0){
  mc_data_row_col<dType, rowType, colType> input;
  input.data = data_in.read();
  input.row = _m1*_config.C1_R + _c1*_config.R + _r1;
  input.col = 0;
  data_out.write(input);

  //NoC: R spatial loop (Y-axis)
  _r1++;
  if (_r1 == _config.R){
    _r1 = 0;
    //NoC: C spatial loop (Y-axis)
    _c1++;
    if (_c1 == _config.C1){
      _c1 = 0;
      //NoC: M spatial loop (Y-axis)
      _m1++;
      if (_m1 == _config.M1){
        _m1 = 0;
}}}}
```

An AGEN first checks if there is data in the interconnect channel connected to the GLB. If data is available, it reads it and labels the data with a row and column id before writing it to the output channel. After sending data, the state machine is updated by incrementing loops affecting spatial mapping.

Psum Read/Update AGEN

The Psum Read/Update AGEN block implements a Read AGEN with corresponding architecture as explained earlier, but also implements an Update AGEN. After all psums are sent to

the PE array using the Read AGEN part, the Update AGEN starts retrieving generated psums and writes them back to the GLB. These ”blocks” keep alternating until all computations for a particular psum are complete. The Update AGEN will then sent a shrink command to the Buffet indicating that it can offload psums to DRAM and restarts itself.

6.4.3 Processing Array

The PE_array block consists of a variable amount of sub-blocks depending on the height of the array. Figure 6.15 provides an overview of the PE_array with a height and width of three. Each row of PEs is contained within a row_block making the array height easily scalable. PE computed psums flow from bottom to top row_block allowing for psum accumulation between row_blocks, this is indicated by the red arrows between them.

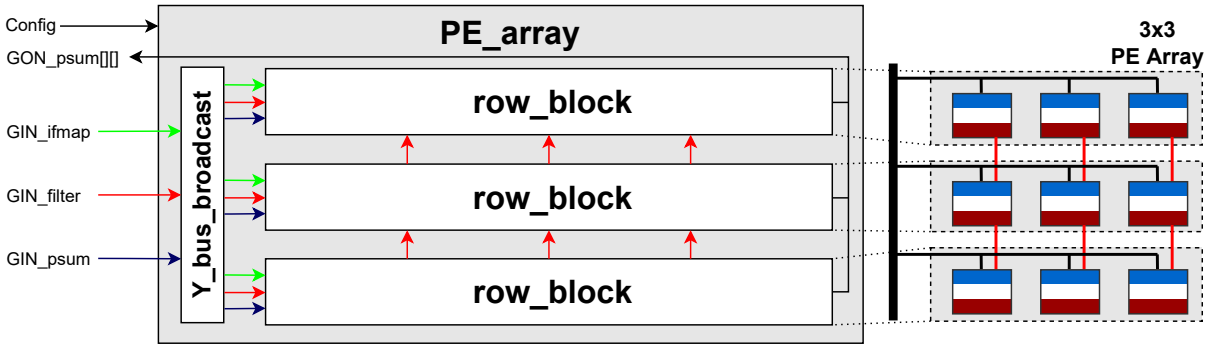


Figure 6.15: Processing Array implementation overview.

Each data type coming from the GLB enters the PE_array via a separate channel. Channels, however, can only have one producer and one consumer. This requires an intermediate block to be added, called the Y_bus_broadcast block, which broadcasts the received ifmap, filter, and psum to all row_blocks.

Row block

The Row_block implements a row of PEs with corresponding Y-bus MCs for each data type. Figure 6.16 shows the implementation overview. Data entering the block first goes through a MC which checks if the pre-configured id matches the row id of the data before it is passed. After data is passed, it is broadcasted via the X_bus_broadcast block to all PE_blocks. The PE_block embodies the PE itself and other blocks explained in the next section.

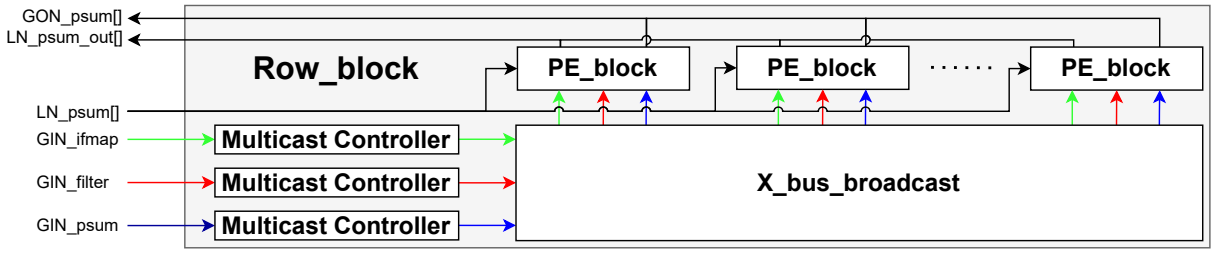


Figure 6.16: Row_block implementation overview.

The number of PE_blocks is made configurable at design time by the Configurator tool. This allows, next to a configurable array height, also, the width to be configurable.

Processing Element block

Each PE is embodied in a PE_block that controls its data inputs with a MC for each data type, see Figure 6.17. Incoming data is filtered by MCs that check if the column id matches with a pre-configured one. Psums, however, are filtered a second time by a Psum In Controller block which is configured to either pass the psum from the GIN or the LN. Computed psums coming out of the PE entering the Psum Out Controller, are either sent to the GON_psum or LN_psum_out bus depending on pre-configuration.

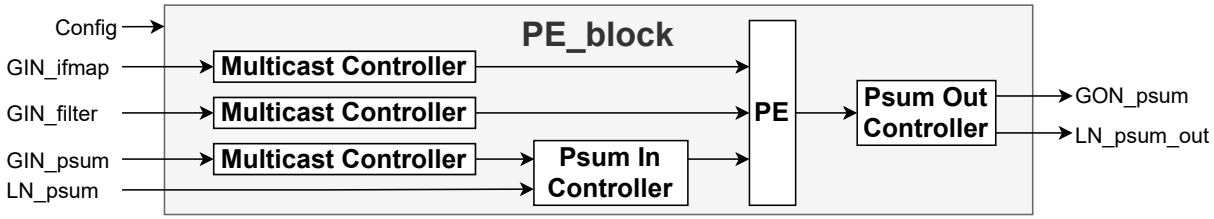


Figure 6.17: Processing Element Block implementation overview.

Processing Element

PEs perform the actual MAC operations and introduce an additional level of storage for each data type. Figure 6.18 shows an overview of the implementation. It implements the inner three loops of the RS dataflow, i.e., the SPad loops. Received ifmaps and filters are directly stored in SPads if possible. Stored psums are accumulated either with the multiplication result of an ifmap and filter or with a received psum.

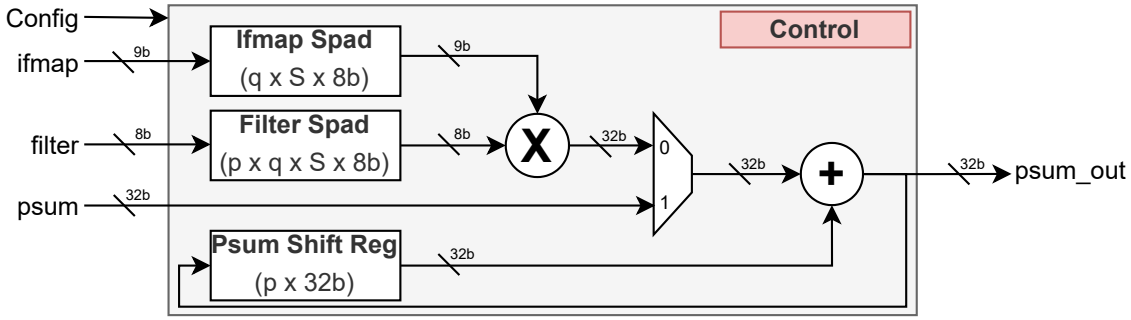
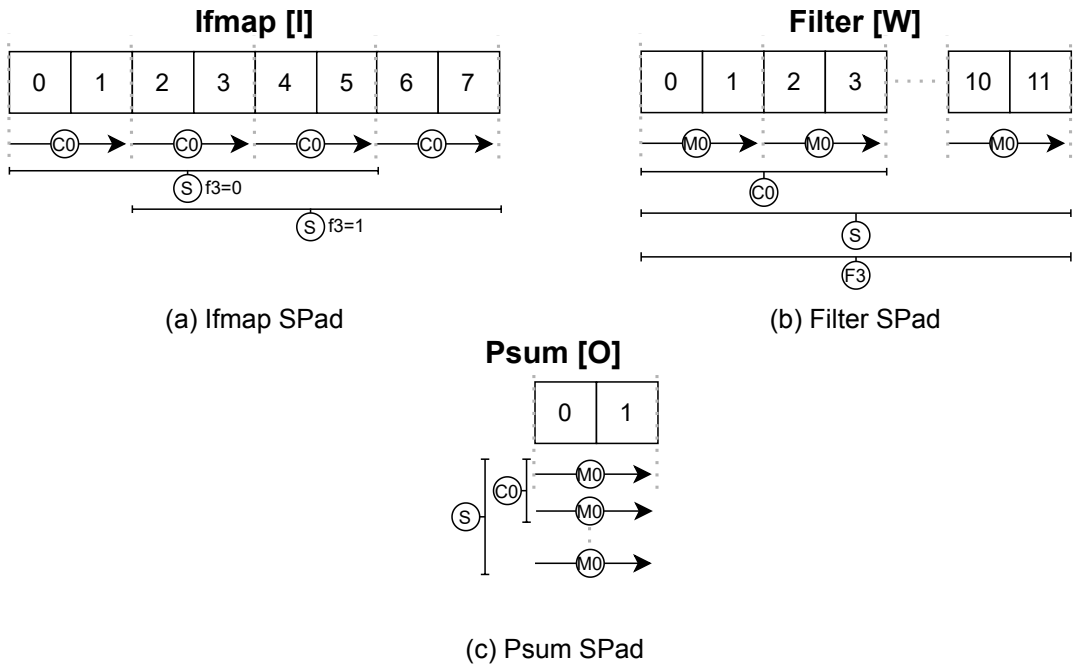


Figure 6.18: Processing Element implementation overview.

Data access patterns and the lifetime of data within a storage element differ between data types. Figure 6.19 illustrates data access patterns and lifetimes of each data type separately. Ifmaps, for example, remain stationary within the inner loop resulting in $m0$ reuses of a single value (Figure 6.19a). Since ifmaps are shifted in using a window, data lifetime depends on $C0$ and stride U. After looping through all SPad loops, $C0 \cdot U$ ifmaps are removed from the buffer in FIFO principle.

Filters remain stationary in a PE when computing a row of psums and are reused F3 times within a PE see Figure 6.19b. Psums get reused $S \cdot C0$ times and storing them can be easily implemented using shift registers.



(a) Ifmap SPad

(b) Filter SPad

(c) Psum SPad

Figure 6.19: Accesses patterns of all storage element in a PE with C0=M0=2 and S=3.

6.5 Configurator

The accelerator is made configurable at design time using the custom-made Configurator tool. This tool allows users to create the accelerator with custom properties. The following properties of the accelerator are made configurable:

- PE array width and height;
- PE SPad sizes for ifmaps, filters, and psums;
- GLB storage capacity for ifmaps, filters, and psum.

The Configurator also parses the generated mappings from all convolutional layers of a model generated by Timeloop. It then creates for each mapping a separate C array containing the corresponding mapping that the accelerator understands. Figure 6.20 shows the Configurator tool with all input files in white and the files it generates in gray.

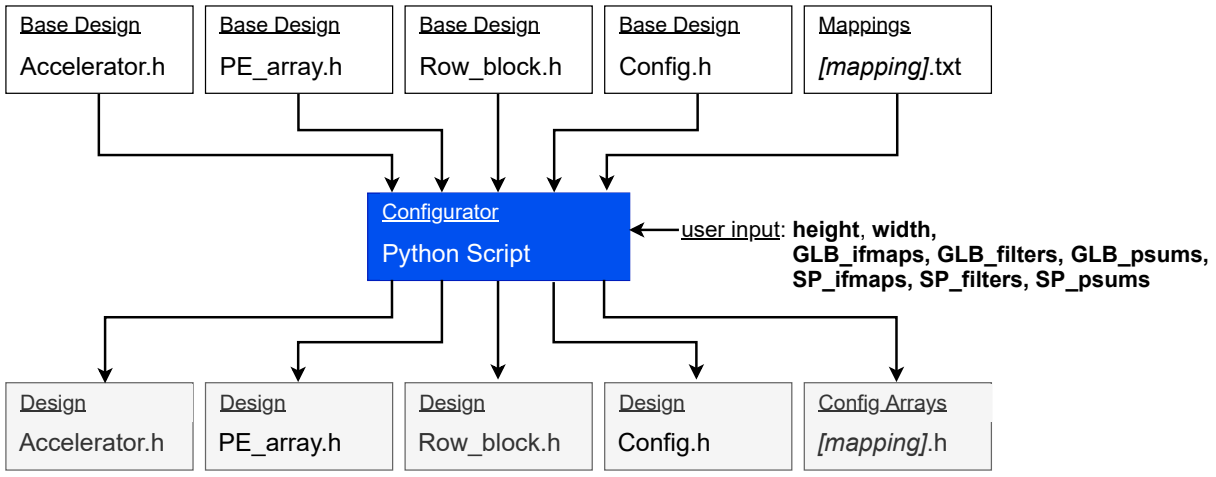


Figure 6.20: Configurator tool overview.

Input files consist, next to mappings, of base design files. Base design files contain the C++ implementation of corresponding blocks with *#pragma*s indicating which parts need to be configured and how. As discussed earlier, the number of Row_block (Row_block.h) instances the PE_array (PE_array.h) creates defines the height of the array. The width of the array is defined by the number of PE_block instances in the row_block. All parameters need to be set in the top block, i.e., set in the Accelerator (Accelerator.h), and how the configuration arrays should be parsed in the Config block (Config.h).

Users must provide the height, width, SPad parameters, and finally, the GLB storage sizes for each data type. After providing the input and starting the Configurator, it generates all corresponding design files and configuration arrays.

6.6 System Integration

The accelerator is connected to the PS via the Advanced Microcontroller Bus Architecture (AMBA). Interfacing the accelerator with DRAM and the CPU is implemented using AXI4 masters and slaves respectively. All Fill AGENs in the Top-Level Control block get connected to an AXI4 master with a configurable base address via a memory-mapped AXI4 slave interface. The CPU configures each master with the base address of the data type it is responsible for. Next to the Fill AGENs, the output shift and output multiplier also get an AXI4 master interface. The accelerator receives its configuration via a memory-mapped AXI4 slave interface. Figure 6.21 shows how the accelerator is integrated with the PS.

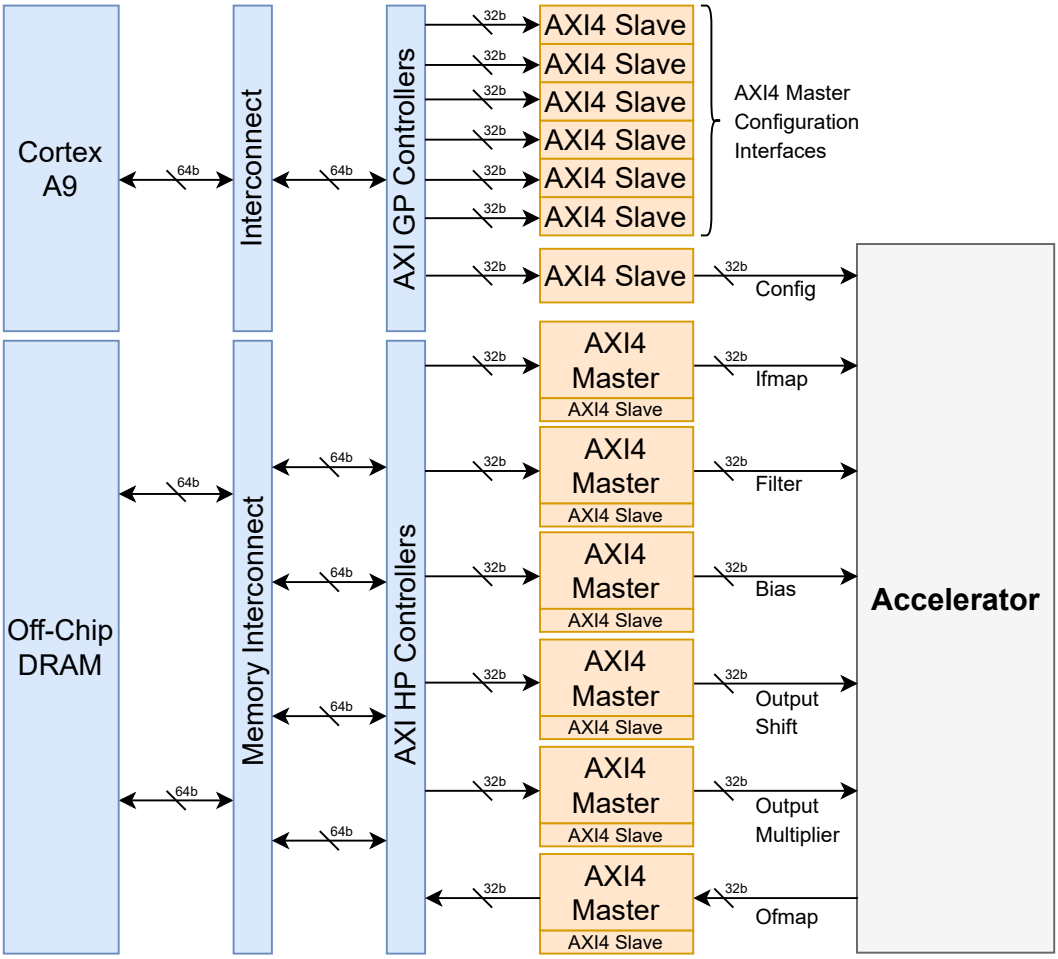


Figure 6.21: System integration overview.

The AXI4 masters are connected to the AXI_HP interface, i.e., AXI HP Controllers, which implement four high-performance, high bandwidth slave ports in the PL that become master ports on the PS AXI interconnect to DRAM [48]. Slaves use the AXI_GP memory-mapped interface, which implements two general-purpose ports in the AXI GP Controllers block. This block is then connected via an interconnect to the CPU.

Note that the AXI4 interfaces, blocks indicated in orange in Figure 6.21, are synthesized by Catapult when arrays get mapped to interfaces. The blue blocks are inferred in Vivado when connecting the accelerator to the PS.

6.7 Catapult High-Level Synthesis Workflow

This section describes the Catapult workflow used to synthesize the HLS C++ design to RTL. Figure 6.22 shows the synthesis tasks, known as the task bar, that controls the workflow. Each task corresponds to a particular stage in the workflow. These tasks advance the design to the corresponding stage by clicking on a task.

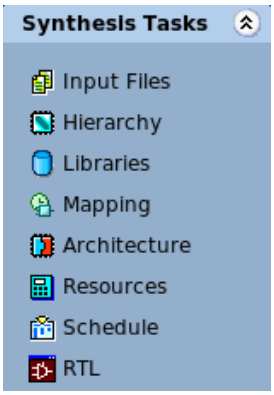


Figure 6.22: Catapult synthesis flow.

The first task is to specify source files to add to the file list of the current solution. Subsequent tasks are elaborated separately below.

6.7.1 Hierarchy

By clicking on the Hierarchy button, Catapult compiles the design. During compilation, files are analyzed and issues reported. It also infers the design hierarchy by identifying the hls_design pragmas. After the task is completed, the Hierarchy Constraint Editor appears, see Figure 6.23. On the left, all source files, including the identified hierarchical blocks are presented. The hierarchy of a block can be configured by clicking on it and setting the desired hierarchy, as shown on the right. Since the Accelerator_3x4 is the top-level block of the entire design and is labeled so using the hls_design top pragma, it is designated Top.

6.7.2 Libraries

After Catapult compiled the design, technology libraries must be specified. Technology libraries contain sets of timing and area estimates for various operators, memories, and registers. Figure 6.24 shows the settings used for this project.

Vivado is set as synthesis tool allowing Catapult to generate script files for project creation with the generated RTL files. The *Compatible Libraries* tab specifies which additional libraries to

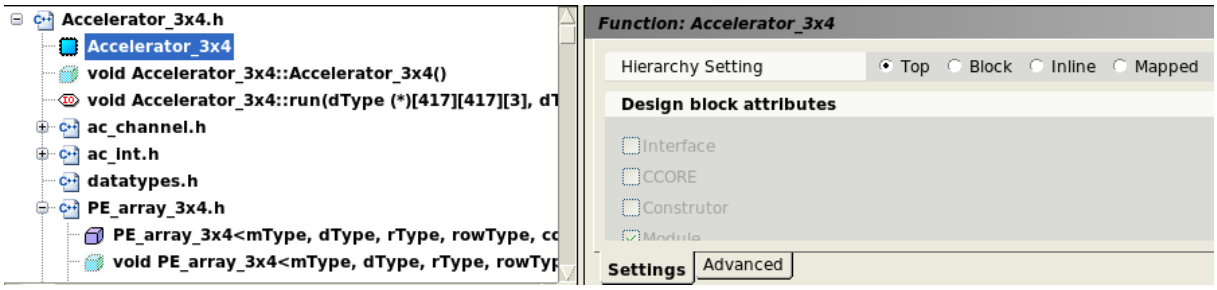


Figure 6.23: The Hierarchy task allows the user to set the hierarchy of blocks.

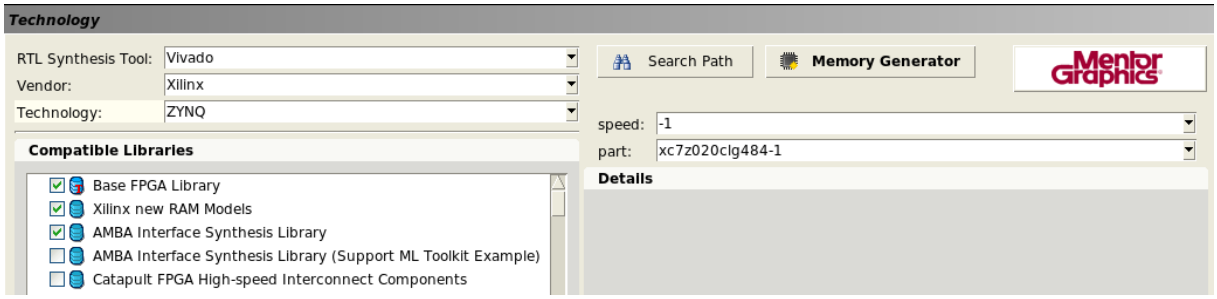


Figure 6.24: Catapult Libraries task.

add next to the base library (*Base FPGA Library*) for the selected technology. Memories in the design too large to be implemented using registers, i.e., GLB and some SPads, must be mapped to RAM blocks of the FPGA. Therefore, the *Xilinx new RAM Models* library is selected. The *AMBA Interface Synthesis Library* contains the implementation of AXI4 blocks which are mapped to the in- and outputs of the accelerator.

6.7.3 Mapping

In the Mapping task, the clock, reset and enable parameters are set. Figure 6.25 shows the settings used for the complete design. The frequency is set to 250 MHz, which is the highest frequency possible on the PL. Other clock parameters are computed automatically.

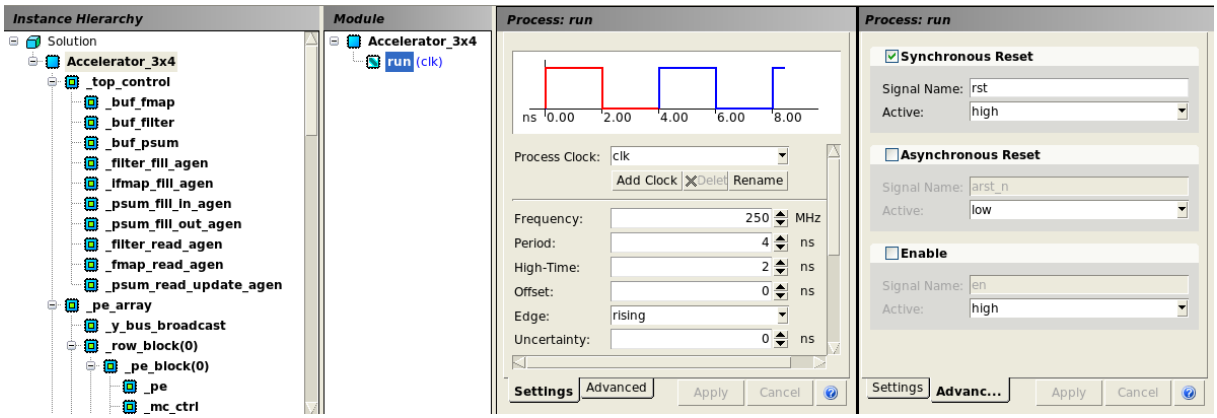


Figure 6.25: Clock, reset and enable parameters are configured in the Mapping task.

The accelerator resets with an active synchronous reset since this is the default for both Catapult and Vivado.

6.7.4 Architecture

After clicking on the Architecture button, Catapult verifies the correctness of interconnects between blocks. It then builds the clock and reset structures, and identifies I/O ports of each block. Now that the entire design has been read into Catapult, each block is evaluated on how it is implemented in the design.

Mapping Interface Resources

Catapult automatically maps the I/O data variables in the source code to input, output, or in-out resources. All port variables and arrays are automatically mapped to separate resources. Arrays defined in the top-level function must be manually mapped to an AXI4 master or slave. Figure 6.26 shows that the filter_in array is mapped to an AXI4 master. All *Resource Options* are set automatically.

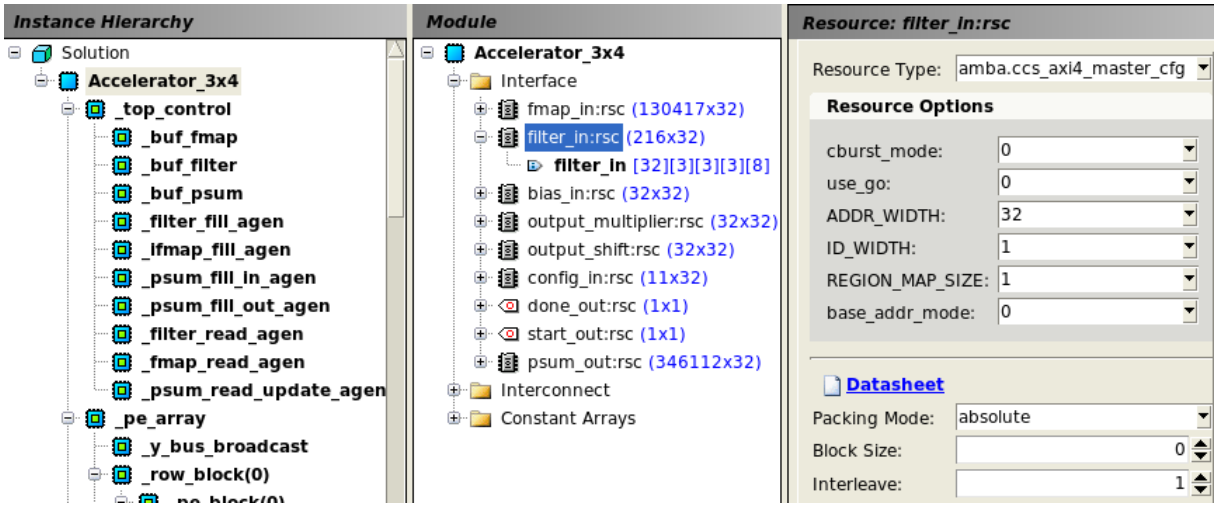


Figure 6.26: Mapping the filter_in array to an AXI4 master interface.

The AXI4 bus data width (Word Width) is set by expanding an interface component and selecting the underlying array definition. Figure 6.27 illustrates this for the filter_in array. The ZedBoard specifications allow for either a 32 or a 64 data width to be selected. Note that, in this case, the word width of the filter_in array is 8, this means that extra logic and storage get introduced by selecting a bigger word width. The advantage of this is that Catapult can reduce bandwidth requirements by fetching or writing multiple values in a single transfer. Since data is mostly accessed non-sequential, the word with is set to 32 bits.

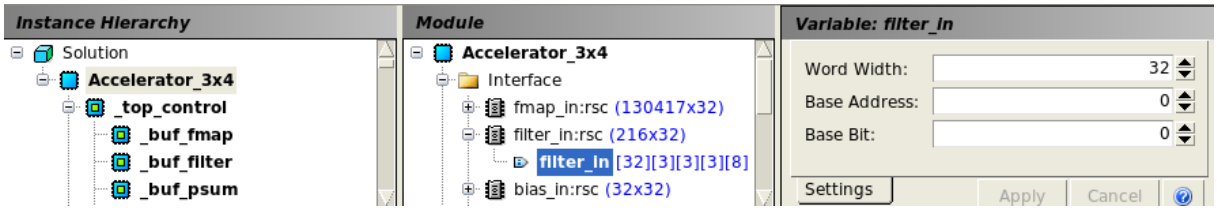


Figure 6.27: Defining the Word With of the filter_in array.

Mapping Memory Resources

Catapult determines automatically how to store memory resources defined in a block, i.e., static or class variables. Figure 6.28 shows, for example, how the SPads, defined as arrays, are

mapped to resources of the FPGA. Because of the low storage requirements of ifmaps, the ifmap array gets mapped to registers. The memories required to store filters and psums, however, are mapped to RAM blocks.

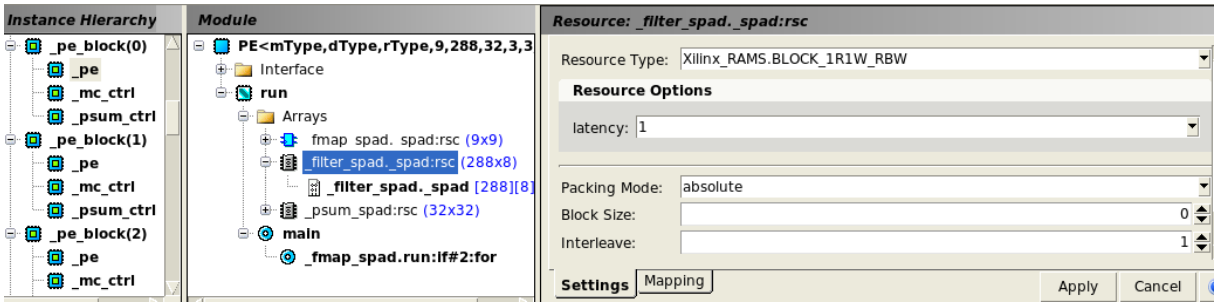


Figure 6.28: Mapping of the filter SPad memory in a PE to a RAM block.

Setting Optimization Constraint

The goal for all blocks in the design is configured to optimize for latency. In Figure 6.29 the optimization configuration for a PE is presented. The effort level is raised from normal to high which forces Catapult to spend up to 10 times more time on scheduling the design resulting in lower latency.

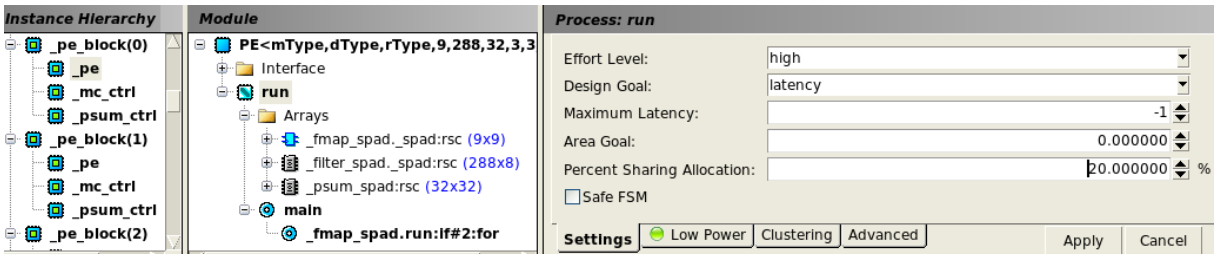


Figure 6.29: Configuring the design goal of a PE.

Configuring Loops

Loops in the design can be unrolled via the Catapult GUI by ticking the *Unroll* box and selecting by how much a loop should be unrolled, see Figure 6.30. Next to unrolling, loops can be pipelined and even entire functions. Figure 6.30 shows that the *main* function is pipelined and that a new ”loop” of the function can start after 4 clock cycles.

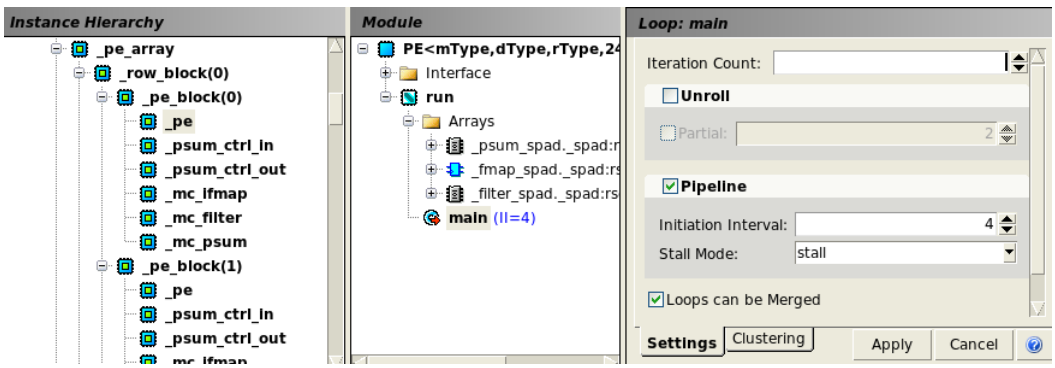


Figure 6.30: Loop unrolling and pipelining configuration.

6.7.5 Resources

By clicking on Resources, Catapult maps resources, such as adders, multipliers, etc., from the design to components available in the *Technology Library*. Mapping resources to components is done automatically and Catapult determines based on the design goal, i.e., latency, area, or power, which component is mapped. Figure 6.31 shows a small part of the automatic mapping result for a PE. On the left side, all blocks with corresponding resource requirements per function are shown. After selecting a resource, the components it can map to appear on the right. By default, the most optimal component is selected, but users can alter this selection.

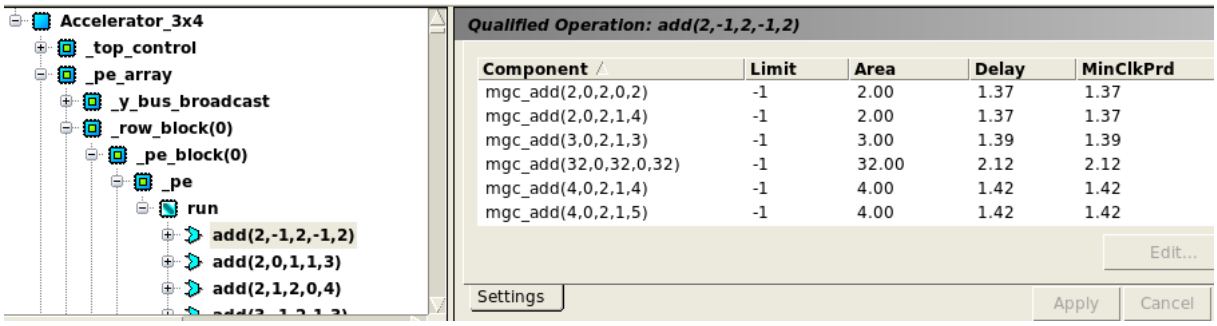


Figure 6.31: Resource to component mapping in the Resources task.

6.7.6 Schedule

After resource allocation, Catapult allows scheduling the design. Catapult applies the timing constraints to the datapath operations and tries to generate a schedule that meets the timing requirements. Figure 6.32 shows the generated schedule as a Gantt Chart for the X_bus_broadcast block.

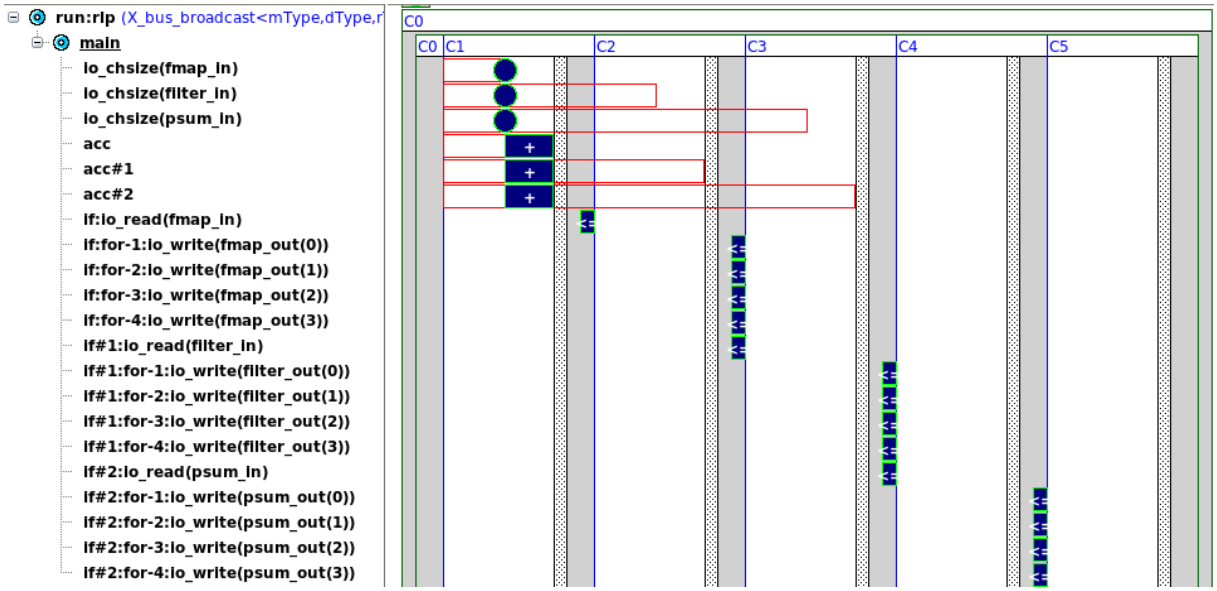


Figure 6.32: Catapult generated schedule for the X_bus_broadcast block.

The Gantt chart graphs the number of control steps (C-steps) in each loop and the sequence of the operations scheduled within the C-steps. C-steps (C0, C1, ... Cn) roughly correspond to states in a finite state machine (FSM). Catapult may map complex conditional statements with several FSM states to a single C-step.

Within each C-step, there is a white, gray, and a shaded area. White and shaded areas comprise the actual clock period. The shaded area represents the percentage of the clock period held in reserve for logic needing to share components and ports. The gray area indicates events that do not affect timing, for example, memory read/write operations. Operations are shown in blue in a box proportional to the operation delay. The red bars around boxes represent slack, indicating that the scheduler is aware that these operations could be scheduled within the complete bar.

Looking back at Figure 6.32, it can be observed that the schedule comprises 5 states, i.e., C1 to C5. In C1, the channel sizes are read indicated by the three circles and function Io_chsize. It then compares the result to see if the output is bigger than 0. This is inferred from the following code:

```
bool available[3];
available[0] = ifmap_in.size()>0;
available[1] = filter_in.size()>0;
available[2] = psum_in.size()>0;
```

If data for a particular type is available, it is read in the next state, i.e., C2 for ifmaps, C3 for filters, and C4 for psums. After reading, the state increments, and it is sent to all outputs of that type. For example, in state C2 ifmaps are read and in state C3 it is sent to all four outputs since this example used a 3x4 PE array. The states for reading and writing are inferred from:

```
if (available[0]){ //Ifmap_in.size()>0
  mc_data_col<mType, colfType> data = ifmap_in.read();
#pragma hls_unroll yes
  for (int i=0; i<LEN; i++){ //LEN = PE array width
    ifmap_out[i].write(data);
  }
}
if (available[1]){ //filter_in.size()>0
  //Logic omitted
}
if (available[2]){ //psum_in.size()>0
  //Logic omitted
}
```

6.7.7 RTL

After generating a schedule that meets the timing requirements, the RTL netlist files can be created by clicking on the RTL task. Catapult generates next to the design files, a script file to launch the generated IP as a project in Vivado, and report files. Table 6.1 gives the generated files used for analysis and final system integration.

Table 6.1: Catapult generated files.

File	File Description
rtl.rpt	RTL design report file
rtl.concat_rtl.vhdl.xv	Script to launch Vivado synthesis tool.
concat_rtl.vhdl	Concatenation of all netlist files for the design into a single HDL.
cycle.rpt	Cycle-accurate design report file

6.8 Post-HLS Design Flow

The last step of the development phase is to integrate the generated RTL of the accelerator into the complete system by connecting it to the PS. For this, the Xilinx Vivado tool is used which is a software suite from Xilinx for synthesis and analysis of HDL designs. The first step in this process, described in Section 6.8.1, is to package the accelerator and export it. This allows it to be easily integrated within other projects. Finally, the accelerator IP module gets integrated into the complete system, see Section 6.8.2.

6.8.1 IP Module Creation

Vivado creates the Catapult generated project for the accelerator by opening the GUI and running the rtl.concat_rtl.vhdl.xv script:

```
source rtl.concat_rtl.vhdl.xv
```

It then auto-generates a project with the accelerator RTL for the Zynq-7020. The IP is then set up to be packaged by using the IP Packager via *Tools* → *Create and Package IP...*. After setup, the packaging steps appear, see Figure 6.33. Most of these steps are performed automatically based on the Catapult generated project settings. The ports and interfaces, i.e., the AXI4 masters and slaves, and done interrupt signal, of the accelerator are, for example, automatically inferred as can be observed in Figure 6.33.

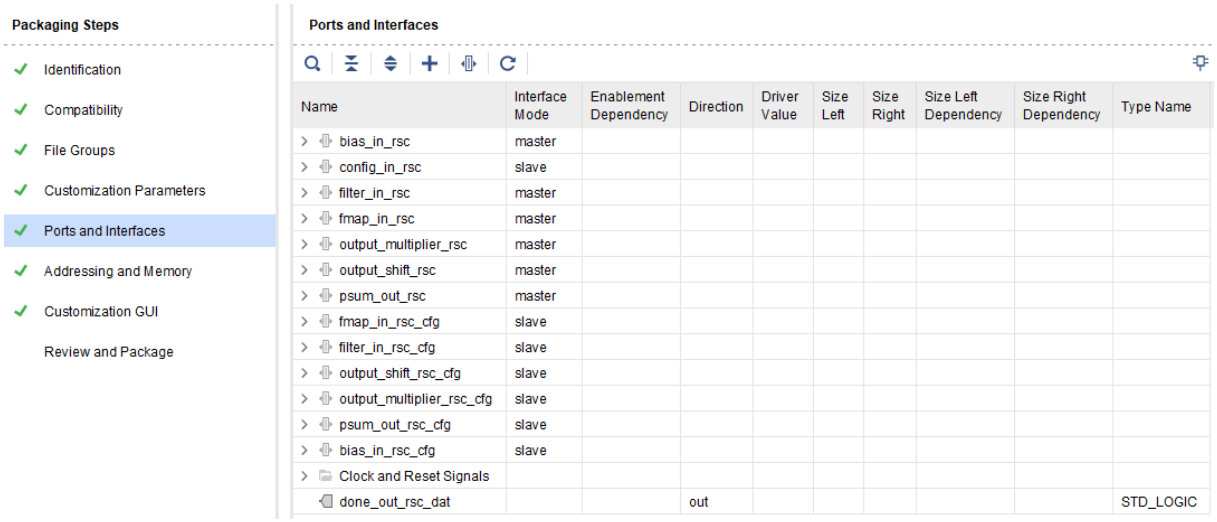


Figure 6.33: Vivado IP *Ports and Interfaces* packaging step.

Finally, in step *Review and Package*, the IP is packaged and exported. The exported IP can then be imported to the IP catalog of another project. Figure 6.34 shows how the exported IP looks like when imported to another project.

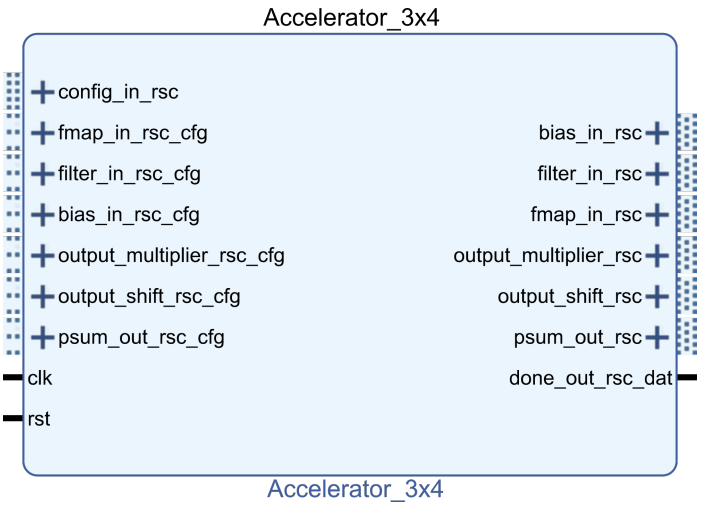


Figure 6.34: Vivado accelerator IP.

6.8.2 System Integration

During system integration, the accelerator gets interfaced to the CPU and DRAM using interconnects. The done_out interrupt signals are connected to the interrupt port of the PS. Figure 6.35 shows the final system integration, with on the right the PS, in the middle, the accelerator surrounded with two interconnects for the two types of busses that are used, i.e., AXI_HP and AXI_GP. Finally, on the left, the Processor System Reset provides customized resets for the entire system.

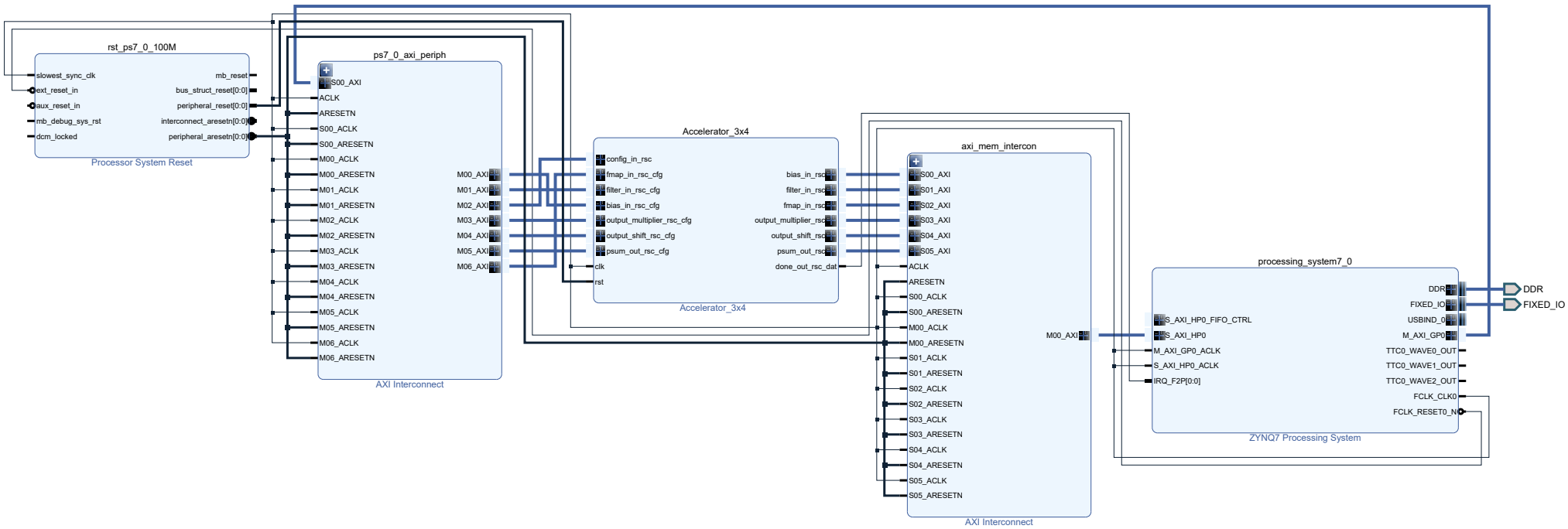


Figure 6.35: Vivado final system integration.

7 EVALUATION AND RESULTS

This chapter presents the results and evaluates them. Results are obtained by running the YOLOv4 tiny model on three different platforms. Unfortunately, due to a lack of time, no results are obtained using the YOLOv4 model. But since the accelerator is model-independent, accelerator validation and performance testing can be done using the tiny model.

This chapter starts by introducing the different platforms used for obtaining results in Section 7.1. Then, Section 7.2 elaborates on the parameters used to configure the accelerator. The resource usage after synthesizing is provided in Section 7.3. Section 7.4 will then present and analyze the performance measured on the three platforms. After obtaining the results, the memory bandwidth is analyzed in Section 7.6, showing potential bottlenecks. Finally, in Section 7.7, the results are compared against those reported in the literature.

7.1 Specifications of Compared Platforms

DNNs normally run on generic hardware platforms such as CPUs and GPUs. It is therefore important to use these platforms as a reference to get a good understanding of how the accelerator affects performance.

7.1.1 Intel Core i7-10750H CPU

The Intel Core i7-10750H is used as the CPU platform. Results are obtained by using the TensorFlow CPU framework. TensorFlow CPU is optimized for SIMD and cache utilization [22]. Table 7.1 presents the specification of the CPU.

Table 7.1: Intel Core i7-10750H CPU specifications.

Cores	Threads	Base Frequency	Max Turbo Frequency	Cache	Max Memory Bandwidth	TDP
6	12	2.60 GHz	5.00 GHz	12 MB	45.8 GB/s	45 W

7.1.2 Nvidia Quadro T1000 GPU

The Nvidia Quadro T1000 is used as the GPU platform. Table 7.2 describes its specifications. TensorFlow GPU allows the models to be executed on the CUDA architecture of the GPU.

Table 7.2: Nvidia Quadro T1000 GPU specifications.

CUDA Cores	GPU Memory	Base Frequency	Max Turbo Frequency	Cache	Max Memory Bandwidth	TDP
896	4 GB GDDR6	1395 MHz	1455 MHz	L1: 64 KB L2: 1024 KB	128 GB/s	50 W

7.1.3 ZedBoard

The software application and hardware accelerator implemented in this work will run on the ZedBoard. The ZedBoard integrates the Zynq-7020 all programmable SoC. Table 7.3 gives the specifications for the PS of the Zynq-7020, and Table 7.4 presents the PL specifications.

Table 7.3: ZedBoard PS specifications.

Cores	Max CPU Frequency	Memory	Max Memory Frequency	Cache	Max Memory Bandwidth
2	667 MHz	512 MB DDR3	534 MHz	L1: 32 KB L2: 512 KB	4.264 GB/s

Table 7.4: ZedBoard PL specifications.

Programmable Logic Cells	LUTs	Flip-Flops	Block RAM (# 36 Kb Blocks)	DSP Slices	Max Memory Frequency
85K	53200	106400	4.9 MB (140)	220	250 MHz

7.2 Accelerator Configuration

The utilization of resources and performance depends on how the accelerator is configured. Table 7.5 presents the parameters used to configure the accelerator.

Table 7.5: Accelerator configuration parameters.

Configuration	Value
PE array height	3
PE array width	4
Ifmap SPad size	24
Filter SPad size	288
Psum SPad size	32
Ifmap GLB size	5000
Filter GLB size	5000
Psum GLB size	10000
PL Frequency (MHz)	125

The height of the PE array is configured to the minimum requirement, which equals the height of the largest filter. The width is set to 4, which allows 4 columns of the output to be computed simultaneously. Unfortunately, the current implementation only supports static mappings, which result in, for example, layers with 13 output columns to be mapped to only a single column, resulting in 25% efficiency regarding columns. The SPad and GLB sizes are configured based on the results obtained from the Eyeriss papers [1][2].

7.3 Resource Utilization

Synthesizing the design for the ZedBoard results in resource utilization, as shown in Table 7.6.

Table 7.6: Resource utilization.

Resource	Utilization	Available	Utilization %
LUT as Logic	42025	53200	78.99
LUT as Memory	1038	1740	5.97
Flip-Flops	56141	106400	52.76
Block RAM (36 Kb Blocks)	65.50	140	46.79
DSP	19	220	8.64

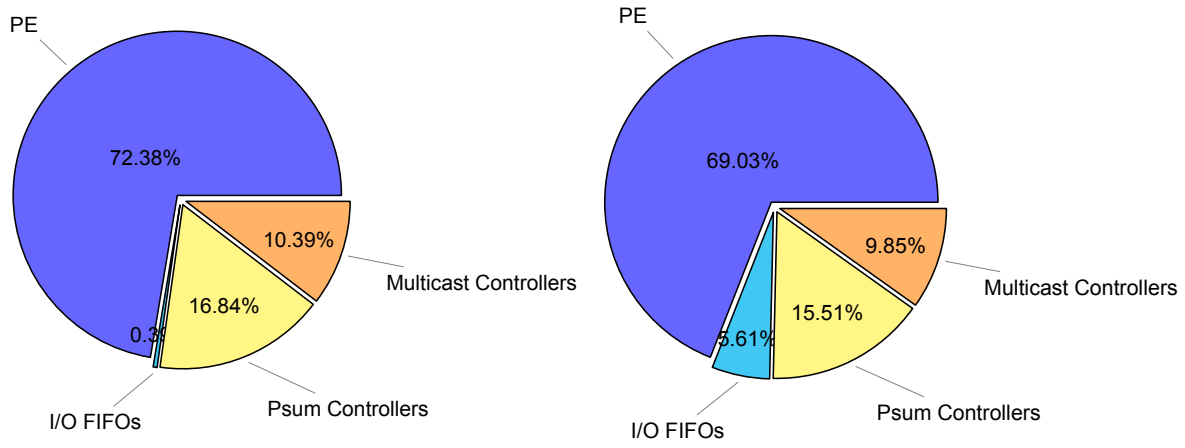
Table 7.7 shows the resource utilization breakdown of each hierarchical top block of the design, AXI4 interconnects connecting the accelerator with the PS, and other components such as leave cells under *Other*. The resource breakdown of a single PE_block is presented in Figure 7.1.

Analyzing the resource utilization breakdown in Table 7.7 shows one potential problem. The main computations of the accelerator consist of MAC operations performed by PEs. Mapping these MAC operations to DSP slices would seem logical since they are optimized to perform such arithmetic operations [48]. However, the total DSPs used by the PE array, as shown in Table 7.7, is zero, i.e., the MAC operations of PEs are not mapped to DSPs. This is due to Catapult not supporting the mapping of arithmetic operations on DSPs for the ARTIX family which the Zynq-7020 is part of. Catapult only supports mapping arithmetic operations on DSPs for the Xilinx VIRTEX-u and VIRTEX-u plus families [7]. The 19 DSPs utilized by the *Top-Level Control* can be explained by Vivado mapping operations to DSPs in the post-HLS design flow.

Table 7.7: Resource utilization breakdown per component.

Component	LUT as Logic	LUT as Memory	BRAM	Flip-Flops	DSP
AXI4 Interconnects	5129	901	11	8410	0
PE array	21177	0	12	29045	0
Top-Level Control	10798	136	6	11821	19
Global Buffer	746	0	20.5	938	0
Config	2673	0	16	3022	0
Other	464	1	0	2905	0

When evaluating Table 7.7 for each component, together with the total utilization of resources given in Table 7.6, configuration improvements can be suggested. First, only 46.79% of the RAM blocks are utilized. This leaves room for increasing the GLB sizes and SPad sizes in PEs that are mapped to RAM blocks. The same applies to arrays mapped to registers. Furthermore, the PE array dimensions may be increased since the resources it uses, i.e., LUT as logic, BRAM, and flip-flops, are still available. Whether these modifications improve the throughput of the accelerator depends on whether the accelerator is currently bandwidth or computational limited.



(a) LUT as Logic.

(b) Flip-Flops.

Figure 7.1: PE_block resource utilization.

7.4 Performance

Performance, as measured on the three different platforms, is given in Table 7.8. The *ZedBoard (PS)* column presents the results of running the models without the accelerator but with the Neon optimized kernels. The last column, i.e., column *ZedBoard (PS+PL)*, gives the results of running the model with the accelerator enabled. Measuring accuracy is performed by using the COCO 2017 validation set [1] containing 4952 images. The accuracy of the ZedBoard generated predictions corresponds to that of the CPU and GPU since it produces bit-accurate predictions compared to that of the original algorithm.

Table 7.8: Performance comparison of running YOLOv4 tiny with an input resolution of 416x416.

Measurement	**CPU**	**GPU**	**ZedBoard (PS)**	**ZedBoard (PS+PL)**
Latency (ms)	89.75	47.89	224625.2	59091.4
Throughput (FPS)	11.14	20.87	0.0045	0.0169
Accuracy (mAP)	0.401	0.401	0.401	0.401
Power (W)	45	50	1.669	2.324
Joule/Image	4.03	2.15	374.89	137.33

The results presented in Table 7.8 show a speedup of 3.84 times on the ZedBoard after enabling the accelerator. Next to the speedup, a 2.73-time reduction in energy per processed image is measured. Unfortunately, the performance and power results of the ZedBoard still lag compared to that of the CPU and GPU. Referring back to the main research question, the results show that it is not possible to create a real-time FPGA design with the Catapult High-Level Synthesis Platform for YOLOv4 on the ZedBoard. Real-time requirements specify that at least a throughput of 30 FPS must be achieved. This requirement is not met since the measured throughput is 0.0169 FPS.

[1] `https://cocodataset.org/#download`

7.5 Analysis

This section analyses the performance as measured and presented in the previous section and compares it against the theoretical maximum achievable performance. Since the accuracy is not affected by the accelerator, only the latency and throughput are analyzed.

This section starts with describing the theoretical analysis principles in Section 7.5.1. Then, in Section 7.5.2, a performance breakdown is presented providing an insight in the obtained results. Finally, the bottleneck affecting the efficiency of the accelerator is elaborated in Section 7.5.2.

7.5.1 Theoretical Analysis Principles

The theoretical throughput of a PE can be expressed in two ways. First, the throughput of a PE can refer to how often, in cycles, a function call can complete [49] indicated by $PE_{throughput}$. The other way is to express the throughput in the number of MACs a PE can perform per second (MAC/s). This depends on its throughput ($PE_{throughput}$) and frequency f. All PEs in this design have a throughput of six cycles. The MAC/s is calculated as:

$$MAC/s = \frac{f}{PE_{throughput}}$$

The workload, i.e., the number of MACs in a convolutional layer, determines the performance upper bound. The performance upper bound is expressed as the time $Proc_{latency}$, in seconds, it takes for the PE array to perform all MACs of a workload, i.e., the optimal processings latency. $Proc_{latency}$ is calculated by dividing the workload by the total MACs that the PE array can perform per second:

$$Proc_{latency} = \frac{Workload}{MAC/s \cdot \#PEs}$$

Finally, the efficiency of the accelerator indicates what percentage of the total time all PEs are active. A 100% efficiency means that all PEs are constantly active. The efficiency is affected when PEs can not perform MACs because of data being absent or a non-optimal mapping. Efficiency is measured between the performance upper bound latency $Proc_{latency}$ and the measured latency $Latency_{meas}$:

$$Efficiency = 100 \cdot \frac{Proc_{latency}}{Latency_{meas}}$$

7.5.2 Performance Breakdown

Table 7.9 provides the performance breakdown per convolutional layer, comparing the latency between the ZedBoard using only the CPU (*PS*) and the ZedBoard with the hardware accelerator enabled (*PS+PL*). The speedup measured between *PS* and *PS+PL* is given in the *Speedup* column. The *#MACs* column describes the workload and column *Latency Bound* the performance upper bound latency. The *Efficiency* column gives the efficiency as measured between the *PS+PL* column and the performance upper bound. Finally, the *PE Array Utilization* column provides the percentage of the PE array being utilized by the mapping of a certain layer. Note that the results in the *Total* row of the *PE Array Utilization* and *Efficiency* columns are compensated for the unsupported layers 18 and 21.

To achieve real-time performance, the accelerator would need to perform 3.31 GMACs in $\frac{1}{30}$ seconds resulting in a throughput requirement of 99.32 GMAC/s. However, the maximum throughput of the PE array clocked at 125 MHz (f), with $PE_{thoughput}$ equal to six cycles, and a total of 12 PEs is 0.25 GMAC/s or 0.072 FPS. This already proves that the accelerator does not meet real-time requirements. Analyzing the results presented in Table 7.9 shows that a maximum theoretical speedup of 16.9x can be achieved, i.e., the total speedup measured between *PS* and *Latency Bound*.

The measured speedup, i.e., the total speedup between *PS* and *PS+PL*, ranges, depending on the layer, from 1.67 times up to 11.67 times. This can be explained by looking at the corresponding PE array utilization and efficiency results. Note that the PE array utilization determines the upper bound of the efficiency. For example, a PE array utilization of 50% causes the efficiency to be 50% or lower. In general, layers with a mapping that utilize more PEs also experience a higher speedup. The efficiency numbers indicate that all PEs are bandwidth limited since all efficiency results are less than their upper bound. This all results in the accelerator being 24.07% efficient. To conclude, the gap between theory and practice is explained by both the PE array utilization and bandwidth limiting the efficiency.

Table 7.9: Performance breakdown of the per convolutional layer.

Layer	PS (ms)	PS+PL (ms)	Speedup	#MACs ($\cdot 10^6$)	Latency Bound (ms)	PE Array Utilization	Efficiency
1	3019.96	258.84	11.67	37.38	149.52	100%	57.77%
2	13570.44	2678.03	5.07	199.36	797.44	100%	29.78%
3	26505.67	3921.85	6.76	393.63	1574.50	100%	40.15%
4	6699.7	969.13	6.91	98.41	393.63	100%	40.62%
5	6699.23	969.37	6.91	98.41	393.63	100%	40.61%
6	3054.53	1446.1	2.11	44.30	177.21	66.67%	12.25%
7	26238.42	3983.88	6.59	388.56	1554.25	100%	39.01%
8	6545.15	984.5	6.65	97.14	388.56	100%	39.47%
9	6543.24	984.88	6.64	97.14	388.56	100%	39.45%
10	2991.62	1443.56	2.07	44.30	177.21	66.67%	12.28%
11	25484	5302.43	4.81	378.54	1514.14	50%	28.56%
12	6395.35	1319.61	4.85	94.63	378.54	50%	28.69%
13	6393.18	1319.94	4.84	94.63	378.54	50%	28.68%
14	2997.36	1445.8	2.07	44.30	177.21	33.33%	12.26%
15	24143.59	13123.21	1.84	358.88	1435.50	25%	10.94%
16	1493.54	895.89	1.67	22.15	88.60	16.67%	9.89%
17	12093.49	6504.55	1.86	179.44	717.75	25%	11.03%
18	1487.75	1486.33	1.00	22.06	88.26	-	-
19	373.78	224.37	1.67	5.54	22.15	16.67%	9.87%
20	38205.54	6140.62	6.22	567.80	2271.22	50%	36.99%
21	2983.17	2982.15	1.00	44.13	176.52	-	-
Total	**223918.71**	**58385.04**	**3.84**	**3310.73**	**13242.94**	**66%**	**24.07%**

7.6 Bandwidth Analysis

It is safe to assume that the current implementation is bottle-necked and, as a result, does not speed up layers significantly. This can be stated since that the original Eyeriss architecture has proven to speed up convolutional layers significantly [1][2].

Analyzing the current bandwidth requirements of the accelerator shows a potential bottleneck that introduces unnecessarily high DRAM accesses. This is illustrated in Table 7.10, which shows the total storage requirements for both ifmaps and filters for each layer and the corresponding DRAM accesses the accelerator performs. The *Accesses/Storage* columns show the average accesses per storage element which should ideally be 0.25 since both ifmaps and filters are 8-bit and each DRAM access fetches 32-bits of data, i.e., each DRAM access can fetch 4 ifmaps or filters. This is, however, impossible since the GLB can not store all fetched data resulting in some data being re-accessed. This potential of fetching 4 values in a single DRAM access is not exploited since data is accessed non-sequentially by the Fill AGENs.

Table 7.10: Ifmap and filter storage and DRAM accesses per convolutional layer.

Layer	Ifmap Storage (MB)	Ifmap DRAM Accesses (MB)	Ifmap Accesses/ Storage	Filter Storage (MB)	Filter DRAM Accesses (MB)	Filter Accesses/ Storage
1	0.5127	2.3419	4.57	0.0009	0.1797	208.00
2	1.3978	25.0399	17.91	0.0184	1.9169	104.00
3	0.6922	32.8008	47.38	0.0369	3.8339	104.00
4	0.3461	8.2002	23.69	0.0092	0.9585	104.00
5	0.3461	8.2002	23.69	0.0092	0.9585	104.00
6	0.6922	11.0756	16.00	0.0041	0.4260	104.00
7	0.3461	32.3748	93.54	0.1475	7.6677	52.00
8	0.1731	8.0937	46.77	0.0369	1.9169	52.00
9	0.1731	8.0937	46.77	0.0369	1.9169	52.00
10	0.3461	11.0756	32.00	0.0164	0.8520	52.00
11	0.1731	42.5984	246.15	0.5898	30.6708	52.00
12	0.0865	10.6496	123.08	0.1475	7.6677	52.00
13	0.0865	10.6496	123.08	0.1475	7.6677	52.00
14	0.1731	11.0756	64.00	0.0655	3.4079	52.00
15	0.0865	63.0456	728.62	2.3593	122.6834	52.00
16	0.0865	5.5378	64.00	0.1311	6.8157	52.00
17	0.0433	31.5228	728.62	1.1796	61.3417	52.00
18	0.0865	5.1917	60.00	0.1306	6.7891	52.00
19	0.0433	1.3844	32.00	0.0328	1.7039	52.00
20	0.2596	31.9488	123.08	0.8847	46.0063	52.00
21	0.1731	10.3834	60.00	0.0065	0.3395	52.00
Total	**6.3234**	**371.2840**	**58.72**	**5.9911**	**315.7207**	**52.70**

The re-accessing of data is better explained by analyzing corresponding Fill AGENs as shown in Figure 7.2a. Loops contained within red boxes influence which index of the data array is being fetched. The blue boxes do not affect this but cause inner loops to access already fetched data. This causes, for example, the Filter Fill AGEN (Figure 7.2b) to re-accesses each filter $4 \cdot E4$ times. Looking back at the result of Table 7.10 and knowing that *E4* is mapped to 52 results in an Access/Storage rate of 208.

Although changing implementation of these parts of the accelerator would lead to an increase in performance, performance can also be increased by creating a more optimal configuration. Increasing the width of the PE array, for example, would reduce *E4* and thereby reduce the Access/Storage rate for filters. The same holds for the Access/Storage rate for ifmaps if the *M4* and *M3* get reduced by increasing the SPad sizes for both filters and psums.

All other potential bottlenecks identified and corresponding improvements are included in the recommendations in Chapter 8.

```
Input Fmaps:      I[N][C][H][W]

//DRAM: storage loops
for (e4=0; e4<E4; e4++){
  for (m4=0; m4<M4; m4++){
    //Global Buffer: storage loops
    for (m3=0; m3<M3; m3++){
      for (c3=0; c3<C3; c3++){
        for (f3=0; f3<F3; f3++){
          //SPad: storage loops
          for (s0=0; s0<S; s0++){
            for (c0=0; c0<C0; c0++){
              //NoC: spatial loops
              for (m1=0; m1<M1; m1++){
                for (c1=0; c1<C1; c1++){
                  for (r1=0; r1<R1; r1++){
                    ifmap = I[0][H][W][C];
}}}}}}}}}}
```

(a) Ifmap Fill AGEN

```
Filter Weights:   W[M][R][S][C]

//DRAM: storage loops
for (e4=0; e4<E4; e4++){
  for (m4=0; m4<M4; m4++){
    //Global Buffer: storage loops
    for (m3=0; m3<M3; m3++){
      for (c3=0; c3<C3; c3++){
        //SPad: storage loops
        for (s0=0; s0<S; s0++){
          for (c0=0; c0<C0; c0++){
            for (m0=0; m0<M0; m0++){
              //NoC: spatial loops
              for (m1=0; m1<M1; m1++){
                for (c1=0; c1<C1; c1++){
                  for (r1=0; r1<R1; r1++){
                    filter = W[M][r1][s0]C];
}}}}}}}}}}
```

(b) Filter Fill AGEN

Figure 7.2: DRAM accesses bottleneck source code analysis.

7.7 Performance Comparison

This section compares the performance of the design presented in this work against the original Eyeriss accelerator and the accelerators previously referred to in Section 3.5. The different accelerators are all compared separately in the sections below. Note that ”this work” refers to the accelerator built in this thesis.

Eyeriss

The developers of Eyeriss [1] made a chip with 168 PEs clocked at 250 MHz resulting in a peak throughput of 42.0 GMAC/s. Comparing the throughput in terms of FPS is difficult since the Eyeriss paper [1] does not reference results obtained using the YOLOv4 or YOLOv4 tiny model. They obtained the results by running the AlexNet [50] and VGG-16 [51] models. This also makes comparing the efficiency difficult since efficiency is model-dependent. However, the throughput in terms of MAC/s can be compared. Each Eyeriss PE has a throughput of 0.25 MAC/s which is about 12 times higher than the PE as implemented in this work. The factor of 12 is explained by the fact that an Eyeriss PE can do one MAC/cycle, compared to $\frac{1}{6}$ MAC/cycle. Combining this with the clock running at twice the frequency results in a speed increase of 12x.

Besides the higher throughput, the minimum PE array utilization is higher at 80%. Mappings for the Eyeriss architecture are created by the Eyeriss mapper, which only targets Eyeriss's row-stationary dataflow. This mapper is not publicly available and is therefore difficult to com-

pare against Timeloop. To summarize, the relatively higher performance of the Eyeriss chip is explained by the following points:

1. Eyeriss PEs have a throughput of one, compared to six of this design.
2. The Eyeriss chip is clocked at double the frequency.
3. The PE array is 14 times bigger with 168 PEs, compared to 12 PEs.

A Demonstration of FPGA-based You Only Look Once version2 (YOLOv2)

The design described in [32] is implemented on the Xilinx Zynq Ultrascale+ MPSoC FPGA which has 5.15x more LUTs and FFs, about 36.19x more BRAM, and 11.45x more DSPs. A throughput of 40.81 FPS and an accuracy of 67.6 mAP is claimed. The higher accuracy is achieved by using the normal YOLOv2 model instead of its tiny version. Comparing the throughput is not possible because no actual throughput number of the PE array is provided and the workload is not stated.

A Power-Efficient Optimizing Framework FPGA Accelerator Based on Winograd for YOLO

Bao [34] presented an accelerator based on the Winograd algorithm. The Winograd algorithm achieves acceleration by reducing the number of multiplications but increasing the number of additions accordingly. Results are obtained by running the YOLOv2 model on the Xilinx PYNQ-z2 board integrating the Zynq 7020-SoC. The author found that quantizing the weights to an 8-bit fixed-point format reduced the accuracy by 8.32%, which is more than measured in this work (4.75%). Because the accuracy decreased so much, it was decided that the final design uses 16-bit precision, reducing the accuracy by only 2.88%.

The claimed accuracy running the YOLOv2 model is 78.25 mAP with a throughput of 8.06 FPS clocked at 125 MHz. Both accuracy and throughput numbers are higher than that of this work. Table 7.11 shows the resource utilization comparison between this work and [34]. The main difference is the number of DSPs utilized. Looking back at Table 7.7 shows that 0 DSPs are utilized by the PE array. Comparing this work with [34] shows that mapping the MAC operation of PEs to DSPs may increase performance and relax resource constraints of the design presented in this thesis.

Table 7.11: Resource utilization comparison between this work and [34].

Resource	This work	Utilization % (This work)	[34]	Utilization % ([34])
LUT as Logic	42025	78.99	38000	71.43
LUT as Memory	1038	5.97	-	-
Flip-Flops	56141	52.76	36000	33.83
Block RAM	65.50	46.79	24.4	17.42
DSP	19	8.64	153	69.55

An FPGA Implementation of Real-time Object Detection with a Thermal Camera

Shimoda [35] introduced a sparse AlexNet-based YOLOv2 model being accelerated by an FPGA-based accelerator. The accelerator achieves a throughput of 164.6 FPS clocked at 100 MHz as implemented on the Xilinx Zynq Ultrascale+ MPSoC FPGA. This relatively high speed up, compared to the design of this and other works, comes mostly from the use of a custom-made model. This model consists of only 8 convolutional layers instead of the 24 convolutional layers of the original YOLOv2 model. The workload is not mentioned, and the accuracy is measured in another metric, which means no comparisons of these variables can be made.

From HLS Component to a Working Design

No exact performance numbers are presented in either the webinar [37] or the corresponding manual [38]. Unfortunately, this means that there are no results to compare against.

Accelerating Tiny YOLO v3 using FPGA-based Hardware/Software Co-Design

Ahmed [36] implemented a completely different architecture based on adder trees that accumulate the multiplications between input and filters, as explained earlier in Section 3.5. The accelerator is implemented on the Virtex-7 VC707 FPGA which has 5.7x more LUTs and FFs, about 7.36x more BRAM, and 16.36x more DSPs. Table 7.12 presents the comparison between the resource utilization of this work and [36]. Incomparable resources are left out indicated by a dashed line. The biggest difference is the number of DSPs utilized. Optimizing the accelerator presented in this thesis to use more DSPs may reduce the *LUT as Logic* resource and thereby allowing an increase of the PE array dimensions.

Table 7.12: Resource utilization comparison between this work and [36].

Resource	This work	Utilization % (This work)	[36]	Utilization % ([36])
LUT as Logic	42025	78.99	48583	16.00
LUT as Memory	1038	5.97	-	-
Flip-Flops	56141	52.76	93225	15.40
Block RAM (36 Kb Blocks)	65.50	46.79	-	-
Block RAM (18 Kb Blocks)	-	-	141	6.80
DSP	19	8.64	2304	82.30

Accuracy numbers are not reported, but the design uses 18-bit fixed-point inputs and filters that would, theoretically, result in a higher accuracy compared to the 8-bit precision of this work. The accelerator achieves a claimed throughput of 460.8 GMAC/s which is 1843.2x higher than this design. However, this throughput is the theoretical upper bound and no real measurements are reported. Since the accelerator is not optimized for data reuse, it is fair to assume that this actual throughput will be lower resulting in low efficiency.

Another disadvantage of the accelerator not being optimized for data reuse is the scalability of the architecture. Increasing the number of PEs would further reduce the efficiency and will, at some point, not cause an increase in throughput since the number of newly added PEs will be completely bandwidth limited. The design in this work, however, is designed to be configurable. This results in PEs with higher data reuse, allowing the number of PEs to be increased without significantly affecting the efficiency.

8 CONCLUSIONS AND RECOMMENDATIONS

This chapter presents the conclusions by answering the research sub-questions and main research questions in Section 8.1. The recommendations for future work are discussed in Section 8.2.

8.1 Conclusions

This section answers all research sub-questions in Section 8.1.1 to Section 8.1.5 and answers the main research question in Section 8.1.6.

8.1.1 Research Sub-Question 1

The first research sub-question is formulated as follows:

Which deep learning framework can be best used for creating the software application?

Literature research presented three suitable frameworks: Darknet [21], TensorFlow [22], and Caffe [23]. After analyzing these frameworks, it was decided to use the TensorFlow sub-framework called TensorFlow Lite Micro (TFLM) [41], as described in Section 5.1.1. The TFLM framework does not require OS support and any standard C or C++ libraries which suit the bare-metal software application it is targeted for. Other benefits of this framework consist of the best tooling, documentation, and support relative to the other frameworks. It allows for the fast development of DNN models in TensorFlow using the Python scripting language which can then be converted to a TFLM compatible model. Another benefit is the SIMD optimized functions that allow layers to execute 5.2 to 20.9 times faster compared to the non-optimized versions.

8.1.2 Research Sub-Question 2

The second research sub-question to be answered is:

Can the YOLOv4 model be optimized before designing a hardware accelerator?

Yes, it has been optimized as follows. The YOLOv4 model is first implemented in Python using the TensorFlow framework. However, the TensorFlow models store their weights as 32-bit floating-point values, which results in complex 32-bit floating-point operations. To reduce the complexity of these operations, weights are quantized from 32-bit floating-point to 8-bit integers using the TensorFlow Lite Converter. This leads to operations being performed in 8-bit and thereby reducing area requirements. An additional benefit is that the size of weights is reduced by a factor of four, resulting in lower bandwidth requirements. The reduction of weight precision decreases the accuracy of the YOLOv4 tiny model, which is used instead of YOLOv4 because of limited time of this project, by 4.75% from 0.421 mAP to 0.401 mAP. Another disadvantage is that the quantization scheme, used to create a correspondence between the bit-representation of values and their interpretation as mathematical real numbers (Section 5.3.1), adds additional complexity to the hardware accelerator.

8.1.3 Research Sub-Question 3

The third research sub-question is formulated as follows:

Which part(s) of the software application can be hardware accelerated?

Computationally intensive functions of the software application are identified by running the YOLOv4 model on the CPU (ARM Cortex-A9) of the ZedBoard. The built-in profiler of TFLM shows that 99.67% of the total execution time of all layers is taken by convolutional layers, see Section 5.2. Analyzing the convolutional algorithm, in Section 5.3, shows that the main computation consists of multiply and accumulates (MACs) which can be highly parallelized. Convolutional layers are therefore well suited to be hardware accelerated.

8.1.4 Research Sub-Question 4

The fourth research sub-question to be answered is:

How can a YOLOv4 accelerator be created using the Catapult High-Level Synthesis Platform?

A proof-of-concept accelerator has been designed that speeds up convolutional layers of the TFLM framework. It is therefore not limited to only speeding up the YOLOv4 model but enables acceleration of all TFLM compatible models.

The accelerator is based on the existing Eyeriss architecture [1][2] that implements the Row Stationary (RS) dataflow, see Chapter 6. The design is modified to integrate the quantization scheme used in TFLM. The implementation of the accelerator complies with the Catapult HLS C++ rules and constraints as presented in Chapter 4. After implementation, the functional correctness of the HLS C++ implementation is tested using a C++ testbench. Catapult supports other verification tools as described in Section 4.6 but no time was left to use these. Synthesizing the HLS C++ design to RTL is done by following the synthesis tasks that correspond to a particular stage in the Catapult synthesis workflow.

Verifying the Catapult generated RTL is performed by testing it with the C++ testbench in Questasim. All necessary files in this process are automatically generated by Catapult. Finally, Catapult generates a Vivado compatible IP which can then be used during system integration described in the post-HLS design flow in Section 6.8.

8.1.5 Research Sub-Question 5

The fifth research sub-question is formulated as follows:

How can the interface between the host PC and the System be implemented?

The interface between the host PC and the system (ZedBoard) is realized by an Ethernet connection using the UDP protocol, as described in Section 5.1.3. The LightWeight IP (lwIP) open-source TCP/IP networking stack from the Xilinx Development Kit provides the implementation of the UDP protocol. This allows the host PC to send images to the ZedBoard and the ZedBoard to send predictions back. Because of the limited time set for this project, the interface is only used for demonstration purposes meaning that the actual execution time of the model is measured on the Zedboard between inference cycles.

8.1.6 Main Research Question

Given the answers to all research sub-questions presented in the previous sections, the main research question can now be answered:

Can a real-time FPGA design be created with the Catapult High-Level Synthesis Platform for the deep learning object detector YOLOv4 on the ZedBoard?

To achieve real-time performance, a minimum throughput of 30 FPS must be achieved. The theoretical performance upper bound of the accelerator shows, described in Section 7.5.2, that even if the accelerator would be 100% efficient, i.e., all PEs are constantly active performing MACs, only have a throughput of 0.25 GMAC/s or 0.0717 FPS is achieved. Increasing the dimensions of the PE array would not lead to a major increase in performance since already 78.99% of most used resource (LUTs) is used. This directly answers the main research question since it shows that the system has a far lower throughput than should be required for real-time performance.

Measurements show that the overall efficiency is 24.07% which further reduces the throughput to 0.0169 FPS. Although the accelerator does not comply with the real-time performance aspects, it speeds up the software application by a factor of 3.84 and decreases energy consumption per processed image by 2.73 times.

Comparing the design with the designs found in the literature shows an important difference in DSP usage. The compared designs map MAC operations to DSPs in contrast to the design presented in this work. As explained in Section 7.3, Catapult does not support the mapping of arithmetic operations to DSPs for the FPGA integrated into the ZedBoard. Utilizing the MAC optimized DSPs [48] may reduce overall resource utilization, allowing the PE array dimensions to be increased.

To conclude, the results show that a real-time FPGA design using the Catapult High-Level Synthesis Platform for the deep learning object detector YOLOv4 on the ZedBoard can not be created. However, this work provides a good foundation for future efforts that can improve the design and may achieve real-time performance by targeting another FPGA with more resources and support for mapping arithmetic to DSPs.

8.2 Recommendations

Features and modifications that are not implemented because of a lack of time are discussed separately in this section.

8.2.1 Processing Element Throughput

Improving the throughput of the PEs would significantly speed up the theoretical performance upper bound. Whether this also applies to the throughput measured in practice depends on the bandwidth limitations of the accelerator. Currently, PEs have a throughput of six cycles. Changing the PE design would allow for a maximum throughput of one cycle, just like the PEs of the original Eyeriss architecture [1]. This provides a maximum theoretical speedup of six times.

8.2.2 Processing Element DSP Mapping

The Zynq-7020 contains 220 DPS slices that can perform different arithmetic operations, including a multiply-accumulator. Currently, the MAC operations of PEs are not mapped to DSPs by Catapult. Mapping these operations to DSPs would be possible since only 8.64% of the total DSPs are currently in use, see Table 7.7. This can reduce the utilization of most resources resulting in more room to increase the PE array dimensions.

8.2.3 DRAM Accesses

As analyzed in Section 7.6, DRAM accesses to fetch ifmaps and filters can be optimized. Reducing the number of accesses to DRAM lowers bandwidth requirements and, thereby, causing a possible speedup. The following points address possible optimizations:

1. Exploit the fact that each DRAM access fetches four ifmaps or filters.
2. Fetch data sequentially from DRAM enabling AXI4 burst transactions and streaming of data.
3. Increase data reuse on the GLB level.

Points 1 and 2 require both modifications on the PL and PS sides. On the PS side, ifmaps and filters should be ordered in memory such that the AGENs of the accelerator can access them sequentially. Doing this for filters is relatively easy since they do not have any dependencies apart from the accelerator, allowing ordering during initialization. This is different for ifmaps because layers executing before convolutional layers are implemented to store their results in a predefined order. Allowing the accelerator to access ifmaps sequentially means that the implementation of these prior layers should also be changed. On the PL side, the fill AGENs are simplified since loops realizing non-sequential data accessing are removed and replaced with a single loop that accesses data sequentially.

Point 3 can be addressed by increasing the GLB storage for ifmaps and filters since only 46.79% of the available BRAM blocks get used.

8.2.4 Dynamic Mapping

The mapping of convolutional layers is currently performed statically, which results in non-optimal mappings. Taking the 3x4 PE array as presented in the results, for example, layers with 13 output columns can only be mapped to a single column of PEs. Dynamic mapping allows the 13 output columns to be mapped in four stages. In the first three stages, the PE array computes the first 12 output columns by splitting them into four separate output columns, which are then computed sequentially by the PE array. In the last stage, only one column of PEs is used to compute the final output column. This modification is, however, not supported by Timeloop and also requires modification to be made in the Configurator and accelerator.

8.2.5 Partial Reconfiguration

The PE array utilization of layers in the YOLOv4 tiny model, as tested on the 3x4 PE array, range from complete utilization (100%) to utilization as low as 16.7%, see Table 7.9. Partial Reconfiguration allows all layers to fully utilize the PE array by creating an optimal PE array for each layer separately. Before executing a layer, the PE array may be reconfigured to most optimally perform the execution of that layer.

8.2.6 GIN Data Bus Width

Currently, the GIN data bus width is limited to one data element, i.e., ifmap, filter, or psum. However, the original Eyeriss architecture [1][2] designed the GIN with a data bus width of four for both filters and psums, i.e., data bus width of $4 \cdot 9b$ and $4 \cdot 32b$, respectively. This may solve potential bandwidth limitation problems within the PE array.

8.2.7 Workload Balancing

During synthesis with Catapult, special care was taken to balance the consumption and production of data between parallel running blocks. However, no time was left to perform detailed analyzes on the design to create an optimal workload balance. Future work should pay attention to this and could balance the workload by pipelining blocks that bottle-neck throughput and or add additional FIFOs between blocks.

8.2.8 Activation Function Integration

The second most time-consuming function of the YOLOv4 model, see Table 5.1, is the PRELU kernel which implements the ReLu activation function. The PRELU kernel is executed after all convolutional layers, except for the convolutional layers directly connected to the output. Integrated the kernel can be best done in the Psum_Fill_Out block since this is where post-processing takes place. This requires, next to the actual ReLu calculation, quantization parameters to be merged of connected convolutional and PRELU layers.

8.2.9 Bare-Metal Multi-Core Application

The ZedBoard comprises two Cortex-A9 cores, however, this project only uses one. Changing the software to a bare-metal multi-core application allows for a maximum speedup of two for the kernels running in software. Xilinx provides the *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors* manual [52] that may help in this process.

8.2.10 Spatial For-Loop *m1*

Unfortunately, the spatial NoC *m1* loop of the dataflow, see Figure 6.4, currently only works in HLS C++ and not in RTL. To bypass this problem, the mapper is not allowed to map over this dimension. Solving this issue would give the mapper more room which may cause mappings with higher utilization.

REFERENCES

[1] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[2] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[4] A. Bochkovskiy, C. Wang, and H.M. Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *ArXiv*, 2020.

[5] Y. Chen, J. Emer, and V. Sze. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro*, 37(3):12–21, 2017.

[6] Siemens. Catapult High-Level Synthesis and Verification. techreport, Siemens Digital Industries Sofware, December 2020.

[7] Mentor. *Catapult Synthesis User and Reference Manual*.

[8] Z. Alom, Tarek T.M, C. Yakopcic, S. Westberg, P. Sidike, S. Nasrin, H. Mahmudul, B.C Van Essen, A.A.S. Awwal, and V.K. Asari. A State-of-the-Art Survey on Deep Learning Theory and Architectures. *Electronics*, 8(3), 2019.

[9] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. *Efficient Processing of Deep Neural Networks*. Morgan & Claypool, 2020.

[10] R. Parhi and R. D. Nowak. The Role of Neural Network Activation Functions. *IEEE Signal Processing Letters*, 27:1779–1783, 2020.

[11] B. Ding, H. Qian, and J. Zhou. Activation functions and their characteristics in deep neural networks. *2018 Chinese Control And Decision Conference (CCDC)*, pages 1836–1841, 2018.

[12] D. Misra. Mish: A Self Regularized Non-Monotonic Neural Activation Function. *CoRR*, abs/1908.08681, 2019.

[13] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[14] D. Gschwend. ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network. mathesis, ETH Zürich, Department of Information Technology and Electrical Engineering, 2016.

[15] S. Bera and V. K. Shrivastava. Effect of pooling strategy on convolutional neural network for classification of hyperspectral remote sensing images. *IET Image Processing*, 14(3):480–486, 2020.

[16] J.L Ba, J.R. Kiros, and G.E. Hinton. Layer Normalization. *ArXiv*, 2016.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[18] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[20] T. Lin, M. Maire, S.J. Belongie, L.D. Bourdev, R.B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C.L. Zitnick. Microsoft COCO: Common Objects in Context. *CoRR*, 2014.

[21] J. Redmon. *Darknet: Open Source Neural Networks in C*. `http://pjreddie.com/darknet/`. Accessed: 2021-01-13.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I.J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D.G Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P.A. Tucker, V. Vanhoucke, V. Vasudevan, F.B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467, 2016.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*, 06 2014.

[24] J. Redmon and A. Farhadi. YOLO9000: Better, Faster, Stronger. *CoRR*, abs/1612.08242, 2016.

[25] J. Redmon and A. Farhadi. YOLOv3: An Incremental Improvement. *ArXiv*, 2018.

[26] TensorFlow. Introduction to Tensors. `https://www.tensorflow.org/guide/tensor`. Accessed: 2021-06-07.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[28] C. Wang, H. Mark Liao, I. Yeh, Y. Wu, P. Chen, and J. Hsieh. CSPNet: A New Backbone that can Enhance Learning Capability of CNN. *CoRR*, abs/1911.11929, 2019.

[29] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia. Path Aggregation Network for Instance Segmentation. *CoRR*, abs/1803.01534, 2018.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *CoRR*, abs/1406.4729, 2014.

[31] S. Woo, J. Park, J. Lee, and I.S. Kweon. CBAM: Convolutional Block Attention Module. *CoRR*, abs/1807.06521, 2018.

[32] H. Nakahara, M. Shimoda, and S. Sato. A Demonstration of FPGA-Based You Only Look Once Version2 (YOLOv2). *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 457–4571, 2018.

[33] H. Nakahara, H. Yonekawa, T. Fujiiand, and S. Sato. A Lightweight YOLOv2: A Binarized CNN with A Parallel Support Vector Regression for an FPGA. *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018.

[34] C. Bao, T. Xie, W. Feng, L. Chang, and C. Yu. A Power-Efficient Optimizing Framework FPGA Accelerator Based on Winograd for YOLO. *IEEE Access*, 8:94307–94317, 2020.

[35] M. Shimoda, Y. Sada, R. Kuramochi, and H. Nakahara. An FPGA Implementation of Real-Time Object Detection with a Thermal Camera. *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 413–414, 2019.

[36] A. Ahmad, M.A. Pasha, and G.J. Raza. Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design. *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020.

[37] Siemens. *From HLS Component to a Working Design*. Siemens. `https://webinars.sw.siemens.com/from-hls-component-to-a-working/room`. Accessed: 2021-06-07.

[38] Xilinx. *Tiny YOLO v2 Machine Learning Design With Catapult Synthesis*.

[39] Post Training Quantization. `https://www.tensorflow.org/lite/performance/post_training_quantization`. Accessed: 2021-05-03.

[40] Model Optimization. `https://www.tensorflow.org/lite/performance/model_optimization`. Accessed: 2021-05-03.

[41] R. David, J. Duke, A. Jain, V.J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *CoRR*, abs/2010.08678, 2020.

[42] ARM Neon. `https://www.arm.com/why-arm/technologies/neon`. Accessed: 2021-05-03.

[43] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A.G. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR*, abs/1712.05877, 2017.

[44] Y. Chen. *Architecture design for highly flexible and energy-efficient deep neural network accelerators*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 2018.

[45] A. Parashar, P. Raina, Y.K. Shao, Y. Chen, V.A. Ying, A. Mukkara, A. Venkatesan, B. Khailany, S.W. Keckler, and J. Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[46] W. Tsai, Y. Lan, Y. Hu, and S. Chen. Networks on Chips: Structure and Design Methodologies. *JECE*, 2012, January 2012.

[47] M. Pellauer, Y.S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S.W. Keckler, C.W. Fletcher, and J. Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–151, 2019.

[48] Siemens. *Zynq-7000 SoC Technical Reference Manual*.

[49] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[50] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*, 25, 01 2012.

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

[52] Xilinx. *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors*.