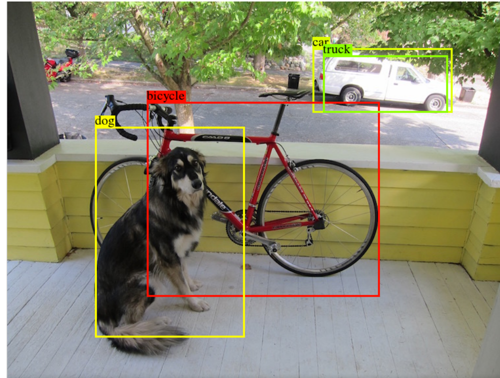
TÉCNICO LISBOA



CGRA-Based Deep Neural Network For Object Identification

Pedro José Runa Miranda

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Horácio Cláudio de Campos Neto
Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Mário Pereira Véstias

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fullfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Resumo

Este trabalho tem como objectivo acelerar uma aplicação para detecção de objectos utilizando a VersatCNN, uma arquitetura de matriz reconfigurável de grão grosso desenvolvida para inferencia de redes neuronais convolucionais (CNNs). O trabalho tem como objectivo final atingir execução em tempo real - 30 imagens por segundo (FPS) - num sistema embebido. A aplicação escolhida para implentar é Tiny YOLOv3, uma CNN desenhada para plataformas com recursos limitados. Para além da inferencia através de uma CNN, a aplicação também inclui funções de pré e pós-processamento. Este trabalho utiliza o IOb-SoC, um sistema num chip (SoC) da IObundle com CPU de arquitetura RISC-V, como plataforma de base para o desenvolvimento do projeto. Ao IOb-SoC são integrados periférios para medição de desempenho e comunicação via ethernet. O desempenho da versão da aplicação em ambientes embebidos é medida para identificar funções a acelerar. A VersatCNN é integrada no sistema como periférico. O trabalho foca-se no desenho de configurações de fluxo de dados distintas para acelerar as diferentes camadas da CNN e as funções de pré e pós-processamento. O sistema final atinge mais de 30 FPS para a aplicação completa, com 143 MHz de frequência de relógio e factor de parallelismo de 832 vezes.

Palavras-chave: Detecção de Objectos, Redes Neuronais Convolucionais, Sistema num Chip, Matrizes Reconfiguráveis de Grão Grosso...

Abstract

The objective of this work is to accelerate an object detection application using the VersatCNN, a Coarse Grained Reconfigurable Array (CGRA) architecture developed for efficient inference of Convolutional Neural Network (CNN). The performance objective is to enable real-time execution - 30 frames per second - on low-end embedded systems. The object detection application accelerated is the Tiny YOLOv3, based on a CNN developed for constrained environments. The Tiny YOLOv3 network is composed of Convolutional, Maxpool, Route, Upsample and Yolo layers. The application also includes pre and post-processing routines. This thesis work uses the IOb-SoC, a RISC-V based system on a chip (SoC) from IObundle as a baseline hardware platform for the project development. Peripheral modules for measuring performance and enabling ethernet communication are integrated into the IOb-SoC. The embedded version of Tiny YOLOv3 is profilled for RISC-V only execution to identify the parts that require acceleration. The VersatCNN is integraded into the system as a peripheral. The work mainly focuses on the design of CGRA dataflow configurations, as the different parts of the application require distinct configuration strategies for the multiple layer types and the pre and post-processing routines. The final system achieves over 30 FPS for the complete Tiny YOLOv3, with a clock frequency of 143 MHz and a 832x parallelism factor for convolutional operations.

Keywords: Object Detection, Convolutional Neural Networks, Systems on Chip, Coarse Grained Reconfigurable Arrays.

Contents

Resumo v
Abstract vii
List of Tables xiii
List of Figures xv
Listing xvii
List of Acronyms xix

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 2
1.3 Thesis Outline 3

2 CNN Background **5**
2.1 Neural Networks 5
2.2 Neural Network Inference 6
2.3 Neural Network Training 6
2.4 Convolutional Neural Networks (CNNs) 7
2.4.1 Convolutional Layer 7
2.4.2 Fully-Connected Layer 7
2.4.3 Pooling Layer 8
2.4.4 Batch Normalization Layer 8
2.4.5 Routing Layer 8
2.4.6 Upsample Layer 8
2.4.7 Shortcut Layer 8
2.5 YOLO 9
2.5.1 Accuracy Metrics 10
2.5.2 Tiny YOLOv3 11
2.6 Conclusions 13

3 CNNs in Reconfigurable Hardware **15**
3.1 General CNN Algorithm 15
3.2 CNN Inference Acceleration in FPGAs 16

3.3 Datapath Optimizations . . . 16
3.3.1 Loop Unrolling . . . 17
3.3.2 Loop Tiling . . . 18
3.3.3 Loop Interchange . . . 18
3.3.4 Design Space Analysis . . . 18
3.3.5 FPGA Implementation . . . 19
3.4 CNN Model Optimization . . . 19
3.4.1 Quantization . . . 20
3.4.2 CNN Model Reduction . . . 20
3.5 Coarse Grained Reconfigurable Arrays . . . 21
3.6 Deep Versat . . . 21
3.6.1 Data Engine . . . 22
3.6.2 Configuration Module . . . 23
3.6.3 Controller and System Integration . . . 23
3.6.4 Deep Versat API . . . 24
3.7 Conclusions . . . 25

4 System Baseline **27**
4.1 Embedded Software Version . . . 27
4.1.1 Batch Normalize Fusion . . . 27
4.1.2 Activation Function Approximations . . . 28
4.1.3 Dynamic Fixed-Point . . . 28
4.1.4 Embedded Software Validation . . . 29
4.2 IOb-SoC Hardware Platform . . . 29
4.2.1 Peripherals . . . 34
4.2.1.1 UART . . . 34
4.2.1.2 Timer . . . 35
4.2.1.3 Ethernet . . . 36
4.3 Tiny YOLOv3 Profiling . . . 37
4.4 Conclusions . . . 37

5 VersatCNN CGRA **39**
5.1 Architecture . . . 39
5.2 Address Generator . . . 41
5.3 Custom Functional Unit . . . 43
5.4 API . . . 44
5.5 Conclusions . . . 46

6 Tiny YOLOv3 with VersatCNN **47**
6.1 Application Overview 47
6.2 Data Storage Format 48
6.3 CNN Acceleration 50
6.3.1 Convolution with Maxpool 50
6.3.1.1 Memory Read Phase Configurations 52
6.3.1.2 Compute Phase Configurations 54
6.3.1.3 Memory Write Phase Configurations 56
6.3.2 Other Convolutional Layers 57
6.3.3 Maxpool Layer 60
6.4 Pre-CNN Processing 61
6.5 Post-CNN Processing 62
6.6 Conclusions 63

7 Results **65**
7.1 IOb-SoC-Yolo System 65
7.2 Tiny YOLOv3 Acceleration 66
7.3 FPGA Resource Utilization 67
7.4 Comparison With Other FPGA Implementations 67
7.5 Conclusions 69

8 Conclusions **71**
8.1 Achievements 72
8.2 Future Work 72

Bibliography **73**

List of Tables

2.1 Yolo layer channel composition and performed activations for $M = 3$. 9
2.2 mAP and inference time comparison between multiple networks, adapted from [7, 23]. . . 11
2.3 Tiny YOLOv3 layers. 12

3.1 Comparison of CNN FPGA implementations, adapted from [25]. 19
3.2 Accuracy comparison with the ImageNet dataset, adapted from [30]. 20
3.3 Accuracy of implementations using CNN model reductions, adapted from [24]. 21

4.1 Linear sigmoid approximation. 28
4.2 Dynamic fixed-point quantization of output FMs, weights and biases of each layer. 29
4.3 mAP_{50} for several Tiny YOLOv3 network models. 29
4.4 IOb-SoC memory mapping. 31
4.5 Timer resolutions and corresponding periods. 36
4.6 Tiny YOLOv3 RISC-V-only performance. 37
4.7 CNN execution time per layer type. 37

5.1 VersatCNN parameters. 41
5.2 Custom FU main functions 43
5.3 Custom FU convolution functions and configurations. 44
5.4 Custom FU configurations for Tiny YOLOv3 layers. 44

6.1 Post-CNN performance using baseline system. 62

7.1 VersatCNN parameters for the IOb-SoC-Yolo implementation. 65
7.2 Tiny YOLOv3 execution times on multiple platforms. 66
7.3 CNN pre and post-processing times. 66
7.4 IOb-SoC-Yolo resource utilization in a Xilinx XCKU040 FPGA. 67
7.5 Performance comparison with other FPGA implementations. 68

List of Figures

2.1 Neural network structure . . . 5
2.2 Plots of common activation functions . . . 6
2.3 Example of a 3D convolution . . . 7
2.4 Diagram of the intersection over union (IoU) calculation, from [21] . . . 10
2.5 Example of precision/recall curve, adapted from [22] . . . 10
2.6 Tiny YOLOv3 layer diagram . . . 11

3.1 Input pixels, kernels and output pixels representation, adapted from [24]. . . . 16
3.2 SIMD accelerator, adapted from [24]. . . . 17
3.3 Spatial and temporal granularity levels, adapted from [35]. . . . 21
3.4 Deep Versat architecture, from [13]. . . . 22
3.5 Versat data engine, adapted from [13]. . . . 23
3.6 Versat configuration module, adapted from [36]. . . . 24
3.7 Deep Versat system, adapted from [13]. . . . 24

4.1 IOb-SoC-Yolo block diagram. . . . 30
4.2 IOb-SoC native interface. . . . 31
4.3 Internal memory block diagram. . . . 32
4.4 Example of an AXI4 read and write transaction (adapted from [40]) . . . 33
4.5 UART data frame. . . . 34
4.6 UART communication protocol. . . . 35
4.7 UART data frame. . . . 36

5.1 VersatCNN detailed block diagram. . . . 40
5.2 Address generator distribution diagram. . . . 40
5.3 Address generator ports. . . . 42
5.4 Class diagram of the VersatCNN CGRA. . . . 45

6.1 Tiny YOLOv3 application diagram. . . . 48
6.2 Data storage formats. . . . 50
6.3 Data tiling diagram. . . . 53
6.4 Data tiling diagram after layer 8. . . . 57

6.5 Upsampling after convolution diagram. 60
6.6 Input and output tiles for the Maxpool layer dataflows. 60
6.7 Pre-CNN resizing process. 61
6.8 Changing label to class colour. 63

Listings

6.1 Embedded Software Program 49
6.2 Convolutional+Maxpool acceleration 51
6.3 Weights and bias configurations for the *Memory Read* phase 52
6.4 IFM tile configurations for the *Memory Read* phase 53
6.5 Weights and bias configurations for the *Compute* phase 54
6.6 IFM tile configurations for the *Compute* phase 55
6.7 Custom FU configurations for the *Compute* phase 55
6.8 vWrite configurations for the *Compute* phase 56
6.9 vWrite configurations for the *Memory Write* phase 56
6.10 Convolutional-only vWrite configurations for the *Memory Write* phase 58
6.11 Custom FU configurations for the Convolutional and Yolo layer pairs 59
6.12 Sigmoid mask calculation 59
6.13 Index pattern and factor calculation 62

List of Acronyms

AGU Address Generator Unit

ALU Arithmetic and Logic Unit

AP Average Precision

API Application Programming Interface

AXI Advanced eXtensible Interface

BRAM Block RAM

CGRA Coarse Grained Reconfigurable Array

CM Configuration Module

CNN Convolutional Neural Network

COCO Common Images in Context

CPU Central Processing Unit

CRC Cyclic Redundant Check

DDR Double Data Rate

DE Data Engine

DFP Dynamic Fixed Point

DGU Data Generation Unit

DMA Direct Memory Access

DSP Digital Signal Processing

FF Flip-Flop

FM Feature Map

FP Fixed-Point

FPGA Field Programmable Gate Array

FPS Frames Per Second

FU Functional Unit

GPP General-Purpose Processor

GPU Graphical Processing Unit

IFM Input FM

IOb-SoC IObundle SoC

IoU Intersection over Union

IPv4 Internet Protocol

LRU Least Recently Used

LUT Look-Up Table

MAC (address) Media Access Control

MAC (block/unit) Multiply-Accumulate

mAP Mean Average Precision

MIG Memory Interface Generator

NMS Non-maximum Suppression

OFM Output FM

PE Processing Element

RAM Random Memory Access

ReLU Rectified Linear Unit

ROM Read Only Memory

SFD Start Frame Delimiter

SFP Static Fixed Point

SIMD Single Instruction Multiple Data

SoC System on a Chip

SRAM Static Random Access Memory

UART Universat Asynchronous Receiver-Transmitter

YOLO You Only Look Once

Chapter 1

Introduction

This thesis describes the acceleration of a compute-intensive application for object detection and classification using a novel embedded system. The embedded system targets low-end integrated circuits. The algorithm uses a Convolutional Neural Network (CNN) approach, which before this work required desktop computers equipped with graphical processors to execute with real-time - 30 frames per second (FPS) - performance.

The innovation of this work is to achieve similar performance in a much simpler and cheaper system. This chapter provides an overview of the motivation and objectives of this work. The chapter also explains the contributions of the author and summarizes the document structure.

1.1 Motivation

Convolutional Neural Networks (CNNs) are the most commonly used methods to develop accurate object detection and classification algorithms. Some networks divide the problem into two stages: proposing regions of interest and performing object localization on each one [1–4]. Other approaches like [5–7] use an end-to-end solution that makes predictions directly from the input image, trading some accuracy for execution speed. As CNN models evolve, they become increasingly more demanding in terms of computation and memory accesses. Graphical Processing Units (GPUs) normally train and deploy CNNs, since they achieve a high number of operations per second. GPUs leverage time parallelism with Single Instruction Multiple Data (SIMD) architectures, and are efficient for processing multiple images (batches). However, GPUs have large form factors, are expensive and consume significant amounts of energy.

There has been a trend to develop dedicated hardware for accelerating CNN inference using reconfigurable hardware architectures such as Field Programmable Gate Arrays (FPGAs). The intent is not only to increase performance but also to lower power consumption.

In these devices, the power consumption is lower and comes mainly from the memory accesses. Besides, FPGAs allow for the application of a range of optimization techniques. Coarse-Grained Field Arrays (CGRAs) enable the same possibilities for exploiting parallelism as in FPGAs but at a higher

level of abstraction. A CGRA is an array of Functional Units (FUs), whereas an FPGA is an array of low-level Look-Up Table (LUT) circuits. The higher level datapath manipulation in CGRAs allows for the creation of toolchains that are simpler for developing accelerators, lowering the barrier of entry for such programmable devices.

1.2 Objectives

The main objective of this work is to develop an implementation of the Tiny YOLOv3 [8] convolutional neural network for an embedded system employing a FPGA or ASIC, capable of achieving a performance of at least 30 frames per second, while detecting and classifying objects in moving pictures or videos. To achieve this objective, a complete Tiny YOLOv3 application is developed and efficiently implemented on VersatCNN, a CGRA architecture geared towards accelerating CNN layers [9]. As such, this work focuses on the design and implementation of the most effective configuration strategies for the VersatCNN that accelerate different CNN layers and other routines.

A SoC platform is used as an hardware baseline for the accelerator deployment. The SoC platform has a RISC-V CPU connected to main memory via a cache system and a peripheral for serial communication. The testing environment requires the integration of an ethernet module to enable large file transfers between a computer and the SoC platform. The SoC platform also requires the addition of a peripheral for performance measurements. The baseline system is used to profile the Tiny YOLOv3 execution and identify target routines to accelerate.

The VersatCNN CGRA is added as a peripheral of the final system. The CPU executes the embedded software and controls the several peripherals, including the VersatCNN configurarions using an C++ Application Programming Interface (API). The CPU also executes parts of the application which have not been hardware accelerated. The different layers of the CNN are mapped into the accelerator core with a set of configuration strategies developed for the embedded software. In addition to the CNN model execution, the application includes functions for pre and post-processing the data. These functions have a significant impact on the global performance of the application. Hence, they also needed to be accelerated.

In this work, all system peripherals were integrated in the IObundle System on Chip (IOb-SoC) platform [10–12], including the VersatCNN accelerator. The other peripherals integrated are a module for performance measurement and an ethernet communication module. The ethernet comunication also required establishing a communication protocol to transfer the input files and output image between a computer and the embedded device. The VersatCNN accelerator was integrated into the final system.

The main contribution of the work consisted in the research and development of very efficient configuration strategies for the execution of the Tiny YOLOv3 functions and CNN layers in the VersatCNN. The strategies for convolutional acceleration maximize the utilization of the computational resources of the CGRA, while avoiding data transfer bottlenecks. The strategies for the rest of the application have lower computational requirements and therefore focus on the data transfer patterns.

1.3 Thesis Outline

This document has the following structure:

- Chapter 2 introduces background knowledge about CNNs and presents the main layers found in CNNs. It introduces the main features of the YOLO [5–7] CNN family, as well as the main accuracy metrics used for evaluation. Finally, the chapter presents the structure of the Tiny YOLOv3 neural network in detail.

- Chapter 3 describes hardware implementations of CNNs. The chapter explores multiple parallelism opportunities in the convolution algorithm, presents results of other works and details the most commonly used model optimization techniques. The chapter finishes with an overview of the Deep Versat [13] CGRA.

- Chapter 4 covers the model optimizations done for embedded deployment and discusses the advantages of hardware implementations. The accuracy of the proposed model is evaluated and compared with the original model. The IOb-SoC [11] platform used as the baseline hardware system is presented along with its main features. The chapter concludes with a performance analysis for the complete Tiny YOLOv3 embedded software version running on the IOb-SoC platform.

- Chapter 5 describes the hardware architecture of the VersatCNN CGRA [9]. It presents the address generator and custom-computing FUs. The chapter closes with a presentation of the VersatCNN API.

- Chapter 6 details the main parts of the Tiny YOLOv3 application. The data format is explained. The configuration for the Convolutional and Maxpool layers is presented in detail. The different strategies used to configure the VersatCNN CGRA for the remaining parts of the application are explained.

- Chapter 7 presents the achieved experimental results for the system developed and compares its performance to desktop CPU and GPU implementations. The chapter concludes by contrasting IOb-SoC with other FPGA implementations of the Tiny YOLOv3 algorithm.

- Chapter 8 provides the final concluding remarks and establishes possible directions for future work.

Chapter 2

CNN Background

Convolutional neural networks (CNNs) were introduced in 1989 by Yann LeCun for digit recognition [14]. Since then, CNNs have increased in size and complexity and have gained popularity in the last years for applications like speech processing, robotics and image processing. One example is the performance of AlexNet [15] at the ImageNet Large Scale Visual Recognition Challenge 2012 [16]. The AlexNet network obtained 37.5% and 17.0% top-1 and top-5 error rates respectively. This result represents a reduction of about 10% in both error rates compared to the winner of the previous competition, and about 8% error rate reduction from the best published results at the time [15].

This chapter establishes a background for CNNs. The first sections introduce the concepts of inference and training of neural networks. The following sections present the main types of layers found in CNNs. The final part of the chapter presents an analysis of the YOLOv3 [7] network, a CNN used for image detection.

2.1 Neural Networks

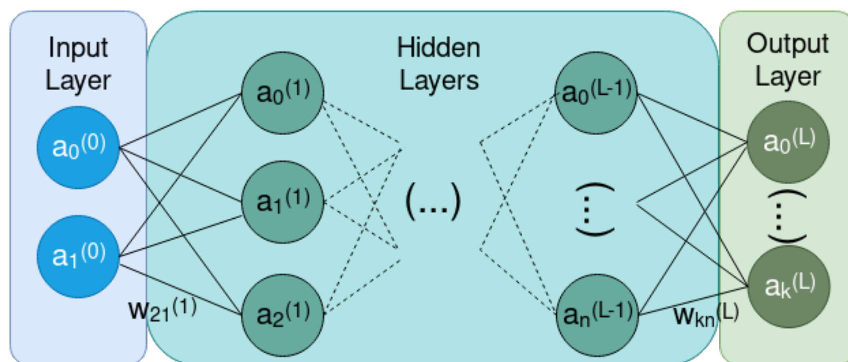


Figure 2.1: Neural network structure

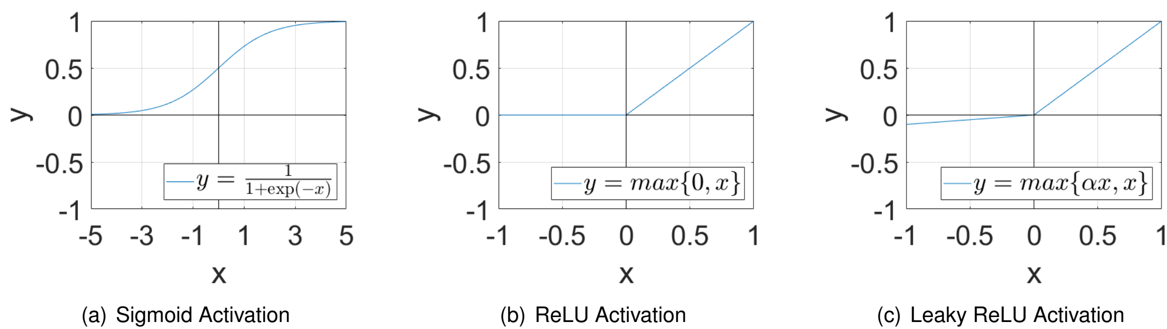
The most basic element of neural networks is the neuron. The output value a of a neuron is given by

$$a = \sigma\Big(b + \sum_{i=0}^{n-1} w_i \times x_i\Big), \tag{2.1}$$

where w_i is the weight associated with input x_i, b is a parameter called bias, $\sigma(.)$ is called the activation function and n is the number of inputs of the neuron.

Figure 2.1 presents a neural network. Multiple neurons form layers where the neurons in one layer only receive input values from the same set of previous layers. If the input values are the neural network inputs, the layer is called the input layer. The neural network sends the values calculated at the input layer neurons to the next layer. The last layer of the network is called the output layer. The layers in between are called hidden layers.

Activations are non-linear functions applied to each neuron output. The non-linearity of the activation functions allows for multiple layer networks to approximate any function [17]. The sigmoid function, presented in figure 2.2(a), is one of the first functions used. In the last years, the Rectified Linear Unit (ReLU), figure 2.2(b), and some variations like the leaky ReLU, figure 2.2(c), have gained popularity due to the reduced computational complexity during network training.



(a) Sigmoid Activation (b) ReLU Activation (c) Leaky ReLU Activation

Figure 2.2: Plots of common activation functions

2.2 Neural Network Inference

Inference corresponds to the operation of the network over data outside of the training dataset using the already trained weights and biases. The key idea behind this mode is that if the input data received is similar to the data used for training, the output generated by the network should reach a comparable accuracy with the one achieved during training.

2.3 Neural Network Training

Neural networks training uses a training dataset that contains inputs for the network and the desired outputs. The training process tunes the weights and biases of the network so that given the dataset inputs, the output generated is as close as possible to the desired output. Training consists of gradient descent techniques that update the weights and biases of the network based on the partial derivatives of each value proportionally to the difference between the desired output and the one generated by the network. The backpropagation algorithm is the main method used for network training. Training

is demanding in terms of computation and memory. The training process is often done in a separate machine, usually taking advantage of the computation capabilities of GPUs.

2.4 Convolutional Neural Networks (CNNs)

CNNs are mainly composed of Convolutional layers. Other commonly found layers in CNNs are fully-connected, pooling, batch normalization, routing, upsample and shortcut layers. These layers are described in more detail in the following subsections.

2.4.1 Convolutional Layer

In Convolutional layers, a neuron output only depends on a small number of inputs, which corresponds to the neuron only analysing a specific feature for a region of the input. The specific region is called the local receptive field. Hence, associated with each neuron, there is only a reduced number of weights and one bias.

In fact, a set of neurons uses the same local weights and bias, which corresponds to searching for the same feature in the entire input, making convolution invariable to translation in images. The shared weights form a kernel.

The input of Convolutional layers is divided into C channels, where each channel is defined as a $(W \times H)$ feature map (FM). For example, in figure 2.3 the input has 3 channels, with each feature map having 6×6 size.

Each kernel used in the convolution has C channels, the same as the input. The number of output channels is the same as the number of kernels used. Each value on the output is the accumulation of the products of the input with the overlapped kernel.

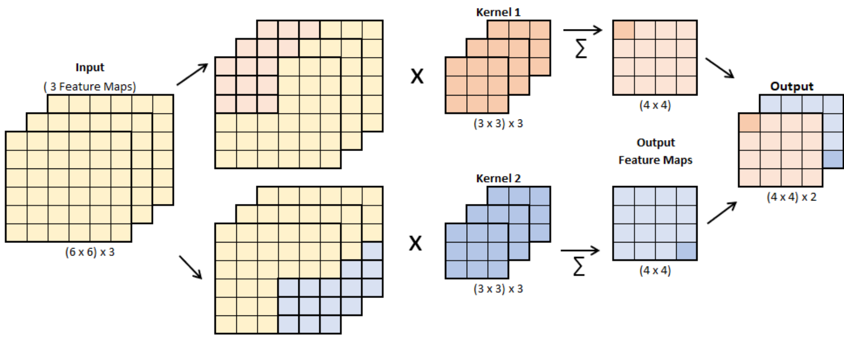


Figure 2.3: Example of a 3D convolution

2.4.2 Fully-Connected Layer

In fully-connected layers, each neuron receives the outputs from all neurons of the previous layer, where there is a different weight associated with each input value. In fact, figure 2.1 depicts a neural network composed exclusively by fully-connected layers.

2.4.3 Pooling Layer

CNNs commonly use pooling layers to reduce the size of the feature maps. Intuitively this corresponds to only keeping a rough idea of the feature location. The reduction of the feature maps also reduces the amount of computation at latter layers of the network. The most common pooling operations divide the feature map into 2×2 regions and select either the larger (Max-pooling) or the average (Average-pooling) value.

2.4.4 Batch Normalization Layer

Batch normalization layers set the average of the input values to zero and the standard deviation to one. After that, the values are scaled and shifted using the (γ,β) parameters also learned in training. This control of the input distribution speeds up training and improves accuracy [18]. For a given value y, the normalization outputs

$$z = \frac{y-\mu}{\sqrt{\sigma^2+\epsilon}}\gamma + \beta, \tag{2.2}$$

where μ is the average, σ is the standard variation, ϵ the scale and β is the batch normalization bias. The additional ϵ term is used to avoid numerical errors related with denominators too close to zero.

Since all variables are known after training, is possible to rewrite (2.2) as

$$z = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}y + \left(-\frac{\mu}{\sqrt{\sigma^2+\epsilon}}\gamma + \beta\right) = Ay + B, \tag{2.3}$$

which avoids the computation of the square root and division.

2.4.5 Routing Layer

Routing layers get an output from a previous layer in the network. Routing layers concatenate the output of all the selected layers.

2.4.6 Upsample Layer

Upsample layers increase the size of the input FMs, increasing their width and height. The simplest upsample method consists in repeating each value in an FM four times in a 2×2 square.

2.4.7 Shortcut Layer

Shortcut layers add the output from two or more previous layers in the network pipeline. Shortcut layers mitigate the vanishing gradient problem that arises during training for networks with considerable depth. With this addition, the networks act as already knowing the result from the further back layer and only need to learn a residual to get the desired outcome. Shortcut layers are common on ResNet networks [19].

2.5 YOLO

YOLO (You Only Look Once) [5–7] is a system for object detection and classification that is composed of a single CNN that receives an image and outputs bounding box coordinates and class probabilities for the detected objects.

A YOLO network is composed of a sequence of Convolutional layers that serve as feature extractors at different scales. Along these layers of the network, either pooling layers or Convolutional layers with stride 2 gradually reduce the size of the FMs, depending on the network version.

After the feature extraction, object detection and classification is done at different FM sizes, which corresponds to analysing the input image at different $g_i \times g_i$ grids.

For each cell in each grid, M attempts of object detection are done. Each attempt is represented by two values for the object box center, two for the object box size, one value for the objectness score and 80 class scores. The objectness score indicates the likelihood of the detection corresponding to an object. The 80 class scores correspond to the number of existing classes in the COCO dataset [20] used for training. In total an object detection attempt is represented by $(2+2+1+80) = 85$ values.

Before the object detection and classification at grid size $g_i \times g_i$ starts, a set of $(M \times 85)$ FMs of size $g_i \times g_i$ is created, and each detection value is associated with a specific FM.

The process of object detection and classification uses a custom type of layer called Yolo layer. At the Yolo layers, the FMs of the channels corresponding to the box coordinates, objectness score and classes scores are passed through a sigmoid activation function (figure 2.2(a)). Table 2.1 presents the channels for each output type and the activation performed at the Yolo layer, for the case of $M = 3$. The outputs of the several Yolo layers correspond to the outputs of the network.

Channel Ranges	Outputs	Activation
$\{[0,1];[85,86];[170,171]\}$	Box center coordinates	Sigmoid
$\{[2,3];[87,88];[172,173]\}$	Box size	None
$\{4;89;174\}$	Objectness score	Sigmoid
$\{[5-84];[90-169];[175-254]\}$	Class scores	Sigmoid

Table 2.1: Yolo layer channel composition and performed activations for $M = 3$.

After obtaining all the outputs of the network, the detections need to be filtered. This consists in excluding detections with an objectness score below a determined threshold; and then applying Non-maximum Suppression (NMS) to eliminate multiple detections of the same object. The NMS process starts from the detection with highest objectness score and resorts to an Intersection over Union (IoU), illustrated in figure 2.4, between the current detection box and all the others. If the result is above the IoU threshold, the detection with the lowest objectness score is discarded, otherwise, is kept. This is done for all remaining detections in descending order of objectness score.

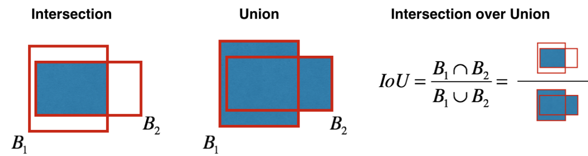


Figure 2.4: Diagram of the intersection over union (IoU) calculation, from [21]

2.5.1 Accuracy Metrics

The mean Average Precision (mAP) is a measure used to compare networks that use the Common Objects in Context (COCO) dataset [20].

The mAP is the average of the AP calculated for each class. The AP is calculated by ordering the classifications performed by the highest bounding box confidence level. The classifications are evaluated as true positives or false positives according to the ground truth. At each classification, the precision and recall are also calculated:

$$recall = \frac{\#True\ Positives}{\#TotalTruePositives} \text{ and } precision = \frac{\#True\ Positives}{\#Classifications}. \tag{2.4}$$

The true positives are the number of correct classifications up to the classification being analysed. The total true positives are the total number of objects in the image. The number of classifications corresponds to the order of the classification being analysed. The orange curve in figure 2.5 presents the precision/recall curve calculated with the values computed for each classification. The curve is then altered to have monotonically decreasing precision, by setting the precision at a given point equal to the maximum precision for any value of higher recall, as is presented by the green curve in figure 2.5. The AP is the area under the green curve.

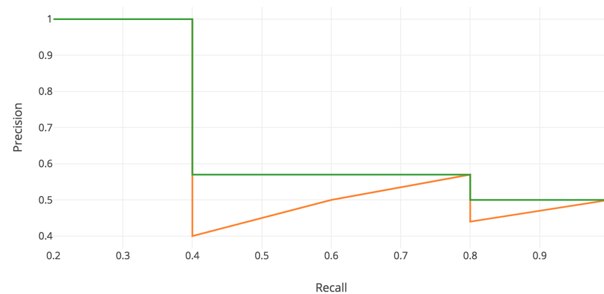


Figure 2.5: Example of precision/recall curve, adapted from [22]

Table 2.2 presents the mAP for a 0.5 IoU threshold and respective inference time (in ms) for multiple networks. The results were obtained using machines with an M40 or Titan X Pascal GPUs [7] as indicated. The results show that the Yolov3 networks achieve accuracy on par with the best performing networks at a fraction of their inference time. The different versions of the Yolov3 networks are related to the resolution of the input images, but all have the same network structure.

Network	mAP_{50}	Time (ms)	FPS	GPU
SSD321	45.4	61	16.4	M40
DSDSD321	46.1	85	11.8	M40
R-FCN	51.9	85	11.8	M40
SSD513	50.4	125	8.0	M40
DSSD513	53.3	156	6.4	M40
FPN FRCN	**59.1**	172	5.8	M40
RetinaNet-50-500	50.9	73	13.7	M40
RetinaNet-101-500	53.1	90	11.1	M40
RetinaNet-101-800	57.5	198	5.1	M40
Yolov3-320	51.5	**22**	**45.5**	Titan X Pascal
Yolov3-416	55.3	29	34.5	Titan X Pascal
Yolov3-608	57.9	51	19.6	Titan X Pascal

Table 2.2: mAP and inference time comparison between multiple networks, adapted from [7, 23].

2.5.2 Tiny YOLOv3

For constrained environments, a small version of YOLOv3 is available. Figure 2.6 presents the diagram of the Tiny YOLOv3 network.

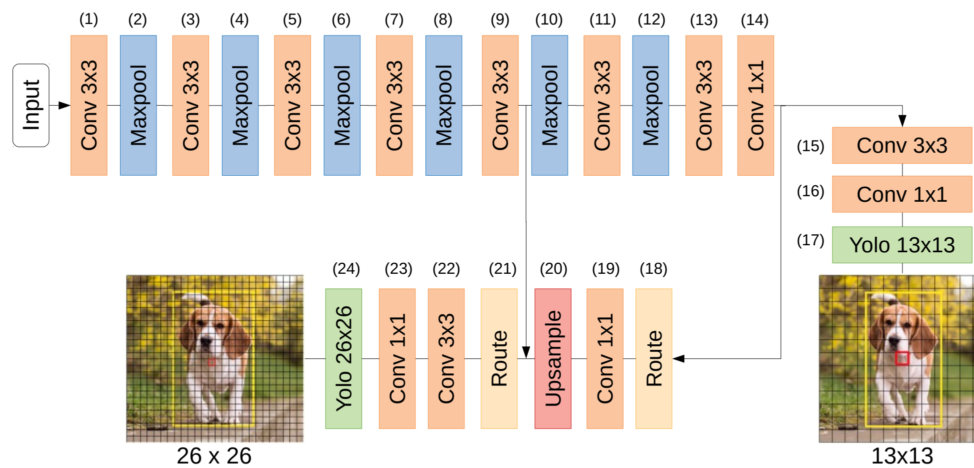


Figure 2.6: Tiny YOLOv3 layer diagram

Table 2.3 details the sequence of layers with regards to the input, output and kernel sizes and the activation function used in each Convolutional layer. Most of the Convolutional layers perform feature extraction. This network uses pooling layers to reduce the FM resolution.

Layer #	Type	Input (WxHxC)	Output (VxUxN)	Kernel (Nx(JxKxC))	Activation
1	Convolutional	416x416x3	416x416x16	16x(3x3x3)	Leaky
2	Maxpool	416x416x16	208x208x16		
3	Convolutional	208x208x16	208x208x32	32x(3x3x16)	Leaky
4	Maxpool	208x208x32	104x104x32		
5	Convolutional	104x104x32	104x104x64	64x(3x3x32)	Leaky
6	Maxpool	104x104x64	52x52x64		
7	Convolutional	52x52x64	52x52x128	128x(3x3x64)	Leaky
8	Maxpool	52x52x128	26x26x128		
9	Convolutional	26x26x128	26x26x256	256x(3x3x128)	Leaky
10	Maxpool	26x26x256	13x13x256		
11	Convolutional	13x13x256	13x13x512	512x(3x3x256)	Leaky
12	Maxpool	13x13x512	13x13x512		
13	Convolutional	13x13x512	13x13x1024	1024x(3x3x512)	Leaky
14	Convolutional	13x13x1024	13x13x256	256x(1x1x1024)	Leaky
15	Convolutional	13x13x256	13x13x512	512x(3x3x256)	Leaky
16	Convolutional	13x13x512	13x13x255	255x(1x1x512)	Linear
17	Yolo	13x13x255	13x13x255		Sigmoid
18	Route	Layer 14	13x13x256		
19	Convolutional	13x13x256	13x13x128	128x(1x1x256)	Leaky
20	Upsample	13x13x128	26x26x128		
21	Route	Layer 9+20	26x26x384		
22	Convolutional	26x26x384	26x26x256	256x(3x3x384)	Leaky
23	Convolutional	26x26x256	26x26x255	255x(1x1x256)	Linear
24	Yolo	26x26x255	26x26x255		Sigmoid

Table 2.3: Tiny YOLOv3 layers.

This network uses two cell grid scales: (13×13) and (26×26). The indicated resolutions are specific to the Tiny YOLOv3-416 version.

The first part of the network is composed of a series of Convolutional and Maxpool layers. Maxpool layers reduce the FMs by a factor of four along the way. Note that layer 12 performs pooling with stride 1, so the input and output resolution is the same. In this network implementation, the convolutions use zero padding around the input FMs, so the size is maintained in the output FMs. This part of the network is responsible for the feature extraction from the input image.

The object detection and classification part of the network performs object detection and classification at (13×13) and (26×26) grid scales.

The detection at a lower resolution is obtained by passing the feature extraction output over 3×3 and 1×1 Convolutional layers and a Yolo layer at the end.

The detection at the higher resolution follows the same procedure but using FMs from two layers of the network. The second detection uses intermediate results from the feature extraction layers concatenated with upscaled FMs used for the lower resolution detection.

The combination of FMs from two different resolutions contributes with more meaningfulness using the information from the upsampled layer and the finer-grained information from the earlier feature maps [7].

2.6 Conclusions

State-of-the-art CNN models are widely used for object detection applications. These networks rely on convolution to extract features from the images to perform object detection. To achieve higher performance metrics, CNNs grow in complexity. The YOLOv3 neural networks present object detection and classification capabilities comparable with other state-of-the-art networks while achieving a higher framerate. The Tiny YOLOv3 network is a reduced model of the Yolov3 network. This network is designed for implementation in devices with constrained resources. The Tiny YOLOv3 application is the CNN model used to demonstrate the acceleration capabilities of the CGRA developed in this work.

Chapter 3

CNNs in Reconfigurable Hardware

CNNs have been implemented on multiple devices, from CPUs and GPUs to FPGAs. CPUs and GPUs mostly use temporal architectures, while FPGAs, which are in the class of reconfigurable hardware devices use spatial architectures to achieve parallelization. Another type of reconfigurable hardware devices is the Coarse-Grained Reconfigurable Array (CGRA) architecture, which, despite powerful, is not as studied as FPGAs for the implementation of CNNs, and is the main focus of this work.

This chapter focusses on the main techniques used to accelerate CNN execution in Reconfigurable Hardware. An effective parallelization is especially important in the convolutional layers which are responsible for 90% of the execution time during inference [24]. Therefore, the chapter starts by highlighting the inherent parallelisms of the convolutional algorithm, followed by a presentation of the main optimization techniques used in previous works. The chapter closes with a review of the Deep Versat CGRA, used in this work.

3.1 General CNN Algorithm

Figure 3.1 presents the data used in a convolution. A particular layer has an input X composed of C FMs of size ($W \times H$) and uses N kernels (Θ_1 to Θ_N) of size ($J \times K \times C$) to obtain the output Y composed of N FMs of size ($V \times U$). Algorithm 1 presents the generic CNN algorithm. The algorithm does not represent the bias for simplicity.

for $n \in \{1, \ldots, N\}$ **do** // Loop 4 iterates over the N channels of output Y
  for $v \in \{1, \ldots, V\}$ **do**
    for $u \in \{1, \ldots, U\}$ **do** // Loop 3 iterates over the pixels within the same output FM
      for $c \in \{1, \ldots, C\}$ **do** // Loop 2 iterates over the channels of input X
        for $k \in \{1, \ldots, K\}$ **do**
          for $j \in \{1, \ldots, J\}$ **do** // Loop 1 iterates over a 2D kernel window
            MAC: $Y[u, v, n] += X[j, k, c] \times \Theta_n[j, k]$

Algorithm 1: General CNN algorithm for a single layer.

The commented loops are candidates for loop unrolling, as discussed in section 3.3.1.

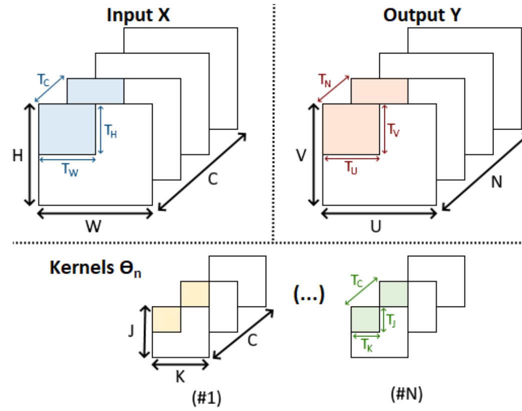


Figure 3.1: Input pixels, kernels and output pixels representation, adapted from [24].

3.2 CNN Inference Acceleration in FPGAs

In general, acceleration for CNN inference in FPGA is done by a combination of datapath and CNN model optimizations.

Datapath optimizations explore parallelism opportunities in the convolutional algorithm described in 3.1. However, only some parallelisms can be explored, due to resource limitations in the devices when compared with the amount of calculations done in a single convolutional layer.

The last optimization type consists of trading model accuracy to improve computational and energy efficiency. This can be achieved by operating over reduced precision operands or reducing the model size.

3.3 Datapath Optimizations

In [24], SIMD accelerators are presented as the main strategy for implementing datapath optimizations. The architecture of a generic SIMD accelerator is presented in figure 3.2. SIMD accelerators receive the input FMs and weights from external memory into an on-chip buffer. This data is then processed in a pool of PEs that output the results into an output buffer. The output is then sent back to the external memory. Each PE contains on-chip registers and performs the computations using DSP blocks. The two levels of caching implemented by the on-chip buffers and registers reduce the accesses to external memory. This reduction has a significant impact in terms of power consumption since the accesses to external memory are the determinant factor for power consumption in FPGAs [18, 24].

With this accelerator architecture, the process of datapath optimization consists in finding the configuration of the PEs and data schedule that maximizes computational throughput, especially for convolutions.

Since the convolution is a set of nested loops (algorithm 1), loop optimization techniques as loop unrolling, loop tiling and loop interchange are applied.

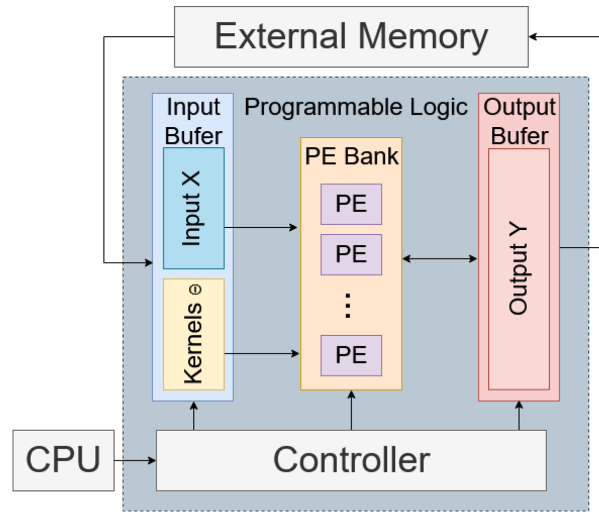


Figure 3.2: SIMD accelerator, adapted from [24].

3.3.1 Loop Unrolling

In algorithm 1, there are four parallelization opportunities, one for each of the commented loops, each one leading to a different dataflow.

- Loop 4, which iterates over the output channels, exhibits Inter-FM parallelism, which allows for parallel calculation of each output FM. For each output FM the same input pixels are used, but a different kernel Θ_n is required. By unrolling loop 4, at each cycle, the same pixel is multiplied by a corresponding weight from different kernels. This leads to uniform access for input pixels and kernels. All computed values belong to pixels in different output FMs, so all of them need to be stored for the next cycle.

- Loop 3, which iterates over the pixels within the same output FM, presents Intra-FM parallelism by calculating multiple output pixels within the same FM. This process uses the same kernel Θ_n, but different input pixels. Unrolling Loop 3 requires accessing different pixels within the same FM but enables the reuse of the weights. All the computed values belong to a different output pixel, so the number of partial values that need to be stored corresponds to the unroll factor applied to this loop.

- Loop 2, which iterates over the input channels, highlights Inter-convolution parallelism as each 3D convolution is a sum of multiple 2D convolutions across the input FMs. Each 2D convolution requires different data from the input and kernel. Unrolling Loop 2 leads to uniform access for input pixels and weights in input FM and 2D kernel coordinates. However, each value pair comes from a different channel. Is possible to add all calculated values into a single partial sum.

- Loop 1, which iterates over a 2D kernel window, reveals Intra-convolution parallelism as each 2D convolution at the two innermost loops can be implemented concurrently. As happens for Loop 2, there is no common data used at each Loop 1 iteration. Unrolling Loop 1 requires accessing different input and weight positions on the same channel. This requires a complex address gener-

ation. All the calculated values are used for the same output pixel and therefore added, requiring the storage of only one partial sum value.

3.3.2 Loop Tiling

Loop tiling is used if the input data in deep CNNs is too large to fit in the on-chip memory at the same time [24]. Loop tiling divides the data into blocks placed in the on-chip memory, like in figure 3.1. The main goal of this technique is to assign the tile size in a way that leverages the data locality of the convolution and minimizes the data transfers from and to external memory. Ideally, each input and weight is only transferred once from external memory to the on-chip buffers.

If the kernel size is bigger than the stride ($J > Stride$ or $K > Stride$), the resulting tiles overlap each other at the boundaries. The added amount of reads is negligible when compared with the tile area [25].

The tiling factors set the lower bound for the size of the on-chip buffer.

3.3.3 Loop Interchange

Loop interchange sets the execution order of the four loops. The innermost loop is the first to be computed, and the outermost loop is the last to be completed.

The loop interchange can be divided into intra-tiling and inter-tiling loop order. The intra-tiling loop order determines the dataflow from the on-chip memory to the PEs, since the data belongs to the same tile loaded into the on-chip buffer. The inter-tiling loop order sets the dataflow from external memory to the on-chip memory, by defining the order of the tiles loaded into the on-chip buffer.

3.3.4 Design Space Analysis

In [25], the loop optimization techniques presented in sections 3.3.1 to 3.3.3 are used to perform an analysis of the SIMD accelerators design space.

The development of the accelerator in [25] is guided by the maximization of the computational parallelism and the minimization of the requirements of partial sum storage, the accesses to the external memory and on-chip buffer.

The computational parallelism has an upper bound determined by the number of multipliers available. However, if the ratios between the loop tiling factors and unroll factors are not integers, the number of multiplications done in parallel are less than the number of available multipliers. The same happens if the ratio between the total data size and the tile size is not integer with regards to the external memory transactions. To avoid the underutilization of the resources, the chosen loop tiling factors should be common factors of the total data size, the same needs to happen between the loop tiling and unrolling factors.

The amount of partial sums that need to be stored depends mainly on the order of loop computations. The reduction of the number of stored partial sums requires the computation of the output pixel values as early as possible. This corresponds to having Loop 1 as the innermost loop.

The accesses to the external memory are tied to the size of the on-chip buffers, which in turn have a size lower bound determined by the loop tiling factors.

Reusing the data sent to the PEs can reduce the accesses to the on-chip buffers. According to the analysis in 3.3.1, this corresponds to unrolling Loop 3 and Loop 4.

3.3.5 FPGA Implementation

In [25], an accelerator architecture is implemented based on the impact of the loop optimizations in performance. The accelerator proposed unrolls loops 3 and 4, which minimizes the on-chip memory accesses.

The tilling is done across W and H of the input data and V and N of the output data. This tiling choice guarantees that the data for Loop 1 and Loop 2 is always buffered and can be computed in sequence. In turn, this reduces the amount of saved partial sums to the product of the unrolling factors. Furthermore, the row tiling at the output requires sequential memory accesses which benefit from DMA data transfers.

Table 3.1 presents a comparison between the works in [25–28], which implement the VGG [29] network in FPGA.

The proposed accelerator in [25] achieves over three times the throughput of the other accelerators. The results highlight the impact in computational throughput of the datapath optimization strategies. For example, between implementations [28] and [25], the same network, at the same frequency achieves over three times the throughput.

Table 3.1: Comparison of CNN FPGA implementations, adapted from [25].

Network [Implementation]	VGG [26]	VGG [27]	VGG [28]	VGG [25]
FPGA	Zynq XC7Z045	Stratix-V GSD8	Virtex-7 VX690t	Arria-10 GX 1150
Frequency (MHz)	150	120	150	150
Number of Weights	50.18 M	138.30 M	138.30 M	138.30 M
DSP Utilization	780 (89%)	1,963[b]	3,600[b]	1,518 (100%)
On-chip RAM[a]	486 (87%)	2,567[b]	1,470[b]	1,900 (70%)
Latency/Image (ms)	224.60	262.90	151.80	47.97
Throughput (GOPS)	136.97	117.80	203.90	645.25
Unrolled Loops	1,2,4	1,2,4	1,2,4	3,4

[a] Xilinx FPGAs in BRAMs (36 Kb) and Altera FPGAs in M20K RAMs (20 Kb)
[b] The resource utilization is not reported in the original papers. The total available resources of the used FPGAs are listed

3.4 CNN Model Optimization

The two most common methods for CNN model optimization in FPGA devices are operand quantization and operation reduction. These methods reduce the number of resources required, enabling more parallelization and reducing data transfers.

CNN model optimizations need to take into account the accuracy loss of the model. To minimize the accuracy loss, these methods are integrated into the model training phase.

3.4.1 Quantization

Quantization is done by reducing the operand bit size. This restricts the operand resolution, affecting the resolution of the computation result. Furthermore, representing the operands in fixed-point instead of floating-point translates into another reduction in terms of required resources for computation.

The simplest quantization method consists of setting all weights and inputs to the same format across all layers of the network. This is referred to as static fixed-point (SFP). However, the intermediate values still need to be bit-wider to prevent further accuracy loss.

In deep networks, there is a significant variation of data ranges across the layers. The inputs tend to have larger values at latter layers, while the weights for the same layers are smaller in comparison. The wide range of values makes the SFP approach inviable since the bit-width needs to expand to accommodate all values.

This problem is addressed by Dynamic Fixed-Point (DFP), which consists of the attribution of different scaling factors to the inputs, weights and outputs of each layer.

In [30], a DFP implementation with 8 bits for the weights and 10 bits for the pixel values, without fine-tuning of the weights, is presented. Table 3.2 presents an accuracy comparison between floating-point and DPF implementations of the same neural networks. The fixed-point precision representation leads to an accuracy loss of less than 1%.

Model Accuracy Comparison	**Single Float Precision**		**Fixed-Point Precision**	
CNN Model	**Top-1**	**Top-5**	**Top-1**	**Top-5**
AlexNet [15]	56.78%	79.72%	55.64%	79.32%
NIN [31]	56.14%	79.32%	55.74%	78.96%

Table 3.2: Accuracy comparison with the ImageNet dataset, adapted from [30].

3.4.2 CNN Model Reduction

Another way to optimize the CNN model is to reduce the number of computations necessary for inference. One method to achieve this is through weight pruning.

Weight pruning consists of eliminating or zeroing the lowest weights. After that, the remaining weights are fine-tuned to improve accuracy. Other criteria for pruning can be developed, like energy-based approaches. In [32] the energy consumption of each layer is estimated by accounting for the accesses to the multiple memory levels and the energy expended for computation. The amount of energy estimated by a layer determines the prunning order of the layers. The more energy-demanding layers are pruned first since the first layers being pruned tend to have better compression ratios than the following layers.

Table 3.3 presents results achieved by CNN model reduction techniques. Both implementations manage to remove over 85% of the weights while keeping over 79% of the original model accuracy. In fact, [33, 34] claim an accuracy reduction within 1% when compared with the respective model using all parameters.

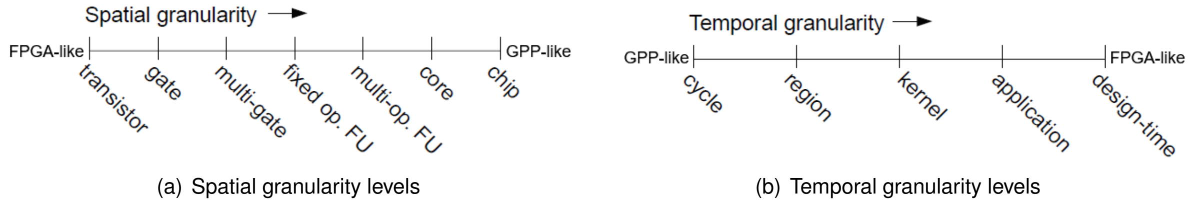
Optimization	Dataset	Removed Parameters (%)	Accuracy (%)	Bitwidth
Prunning [33]	Cifar10	89.3	91.53	8 Fixed-Point
Prunning [34]	ImageNet	85.0	79.70	32 Float

Table 3.3: Accuracy of implementations using CNN model reductions, adapted from [24].

3.5 Coarse Grained Reconfigurable Arrays

Coarse-Grained Reconfigurable Arrays (CGRAs) are presented in [35] as a middle ground between FPGAs and general-purpose processors (GPPs). A tendency from both FPGAs and GPPs towards CGRAs is highlighted: on the one hand, FPGAs tend to have more coarse-grained blocks like DSPs, while GPPs become more heterogeneous to include special instructions and accelerators.

In [35], CGRAs are defined as having temporal granularity at the region or loop-nest level or above (figure 3.3(b)) and spatial granularity at the fixed functional unit (FU) level or above (figure 3.3(a)). As such, CGRAs can change the architecture to fit the application during runtime.



(a) Spatial granularity levels

(b) Temporal granularity levels

Figure 3.3: Spatial and temporal granularity levels, adapted from [35].

CGRAs are integrated into systems mainly as accelerators. The runtime reconfigurability allows for changes in the datapath, which allows for hardware reuse in complex applications. At the same time, the FU granularity enables enough degree of control for applications that do not take advantage of bit-level manipulation.

With an adequate toolchain for assembly, compilation and place and route, CGRAs have the potential to facilitate HW acceleration for multiple application types.

Neural networks are one of such applications. All layers in a neural network have the same elemental operations, which are mostly MACs (multiply-accumulate) and data transfers. Therefore, with tailored FUs and correct datapath configuration, a neural network acceleration environment based on CGRAs seems very attractive.

3.6 Deep Versat

Deep Versat is a CGRA architecture proposed and implemented in [13]. Deep Versat is composed of Versat Layers organized in a ring structure, as presented in figure 3.4.

Each Versat Layer is composed of a Configuration Module (CM) and a Data Engine (DE). This architecture provides a sustainable way to scale the Versat architecture developed in [36]. The scalability comes from the fact that the inputs from a Versat Layer can select data from the same Versat Layer or

the previous one. With this connection policy, the routing complexity is independent of the number of deployed Versat Layers.

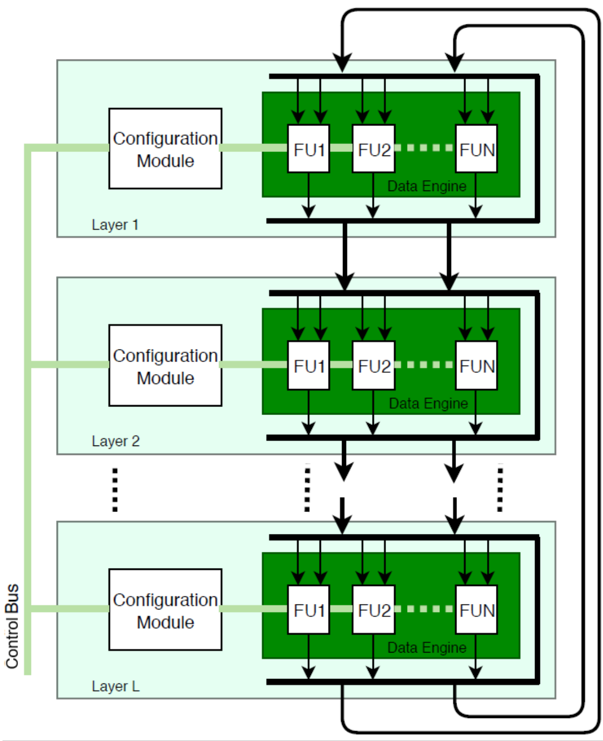


Figure 3.4: Deep Versat architecture, from [13].

3.6.1 Data Engine

The Data Engine (DE) at each Versat Layer is where computations take place. The DE is composed of a set of Functional Units (FUs) in a full-mesh topology. Even though the number of FUs is variable, the topology presents scalability problems, so the author recommends configuring the DE so that each FU input can select one out of a maximum of 20 data sources (10 of then from the previous Versat Layer). Figure 3.5 presents a diagram of a DE.

The DE also contains a Configuration Bus to configure each FU with the type of operation and input selection.

The FUs in a typical DE are Arithmetic and Logic Units (ALUs), multipliers, barrel shifters and dual-port embedded memories.

All FUs have a single output port except for the dual-port memories, which have two. The maximum number of FUs is 10 so that the Versat layer does not produce more than 10 data sources that can be selected by the FUs themselves. This limitation is imposed to tackle the scalability problems of the architecture.

The dual-port memories have two data inputs and outputs and two Address Generation Units (AGUs) that can generate addresses sequences for the two memory ports. The AGUs can generate addresses for variables in nested double loops. Each memory port can read from or write to an address generated

by its AGU, an address input to the other port, or can work as a Data Generation Unit (DGU), outputting the sequence generated by the AGU to the memory port itself.

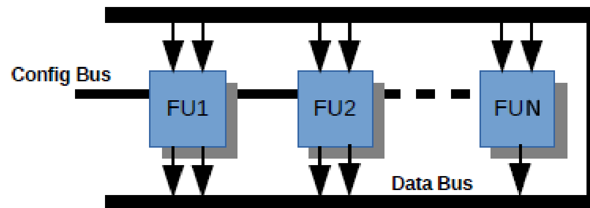


Figure 3.5: Versat data engine, adapted from [13].

3.6.2 Configuration Module

The Configuration Module (CM) is the module where the DE configurations are stored and managed. Figure 3.6 presents the structure of the configuration module. The main components of this module are the configuration memory, register file and shadow register.

The shadow register holds the configuration currently being executed by the DE. The existence of this register allows for changes in the register file without affecting the DE execution.

The configuration register file is composed of configuration spaces that in turn have multiple configuration fields. One configuration field has the control bits for a particular feature of a single FU, while a configuration space has the set of configuration fields for an FU. Each configuration field can vary in bit width as the different FU features require different numbers of configuration bits. Configuration fields exploit time locality as the whole DE configuration changes little over time. Furthermore, by being addressable at the configuration field level, the partial configuration of an FU becomes possible.

The register file takes advantage of time locality, as it is likely that the same DE configuration is valid for a time span, or similar configurations are used in sequence.

The configuration memory can store 64 full DE configurations, the most used ones usually, which can be loaded from or uploaded to the register file or external memory, opening the possibility for more configuration storage in the system.

3.6.3 Controller and System Integration

A soft processor using a RISC-V architecture controls the Deep Versat layers. This architecture is programmable with the GNU standard toolchain, which contains compilers for C and C++.

In [13], Deep Versat is connected to the controller as a peripheral using the system control bus. It has a data interface connected to a data bus. Each bus is memory mapped by a distinct memory base address. There are also dataflow buses that connect two consecutive Versat layers.

The control bus is used to start a Deep Versat run and verify its completion. Additionally, this bus can access the Versat memory FUs and is also used to manage the configuration registers and memory.

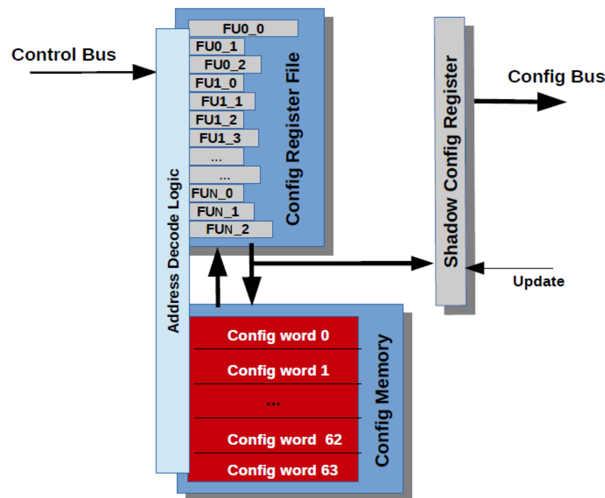


Figure 3.6: Versat configuration module, adapted from [36].

The data bus connects Deep Versat to data sources and sinks and should be used for larger and faster data transfers. The author comments on the inefficiency of driving the data through the RISC-V processor and suggests the implementation of a DMA engine connecting Deep Versat directly to the external memory instead.

Figure 3.4 presents the system architecture, where there is also a UART peripheral, used mainly for printing debug and verification messages. Each Versat block represents a Versat layer of Deep Versat.

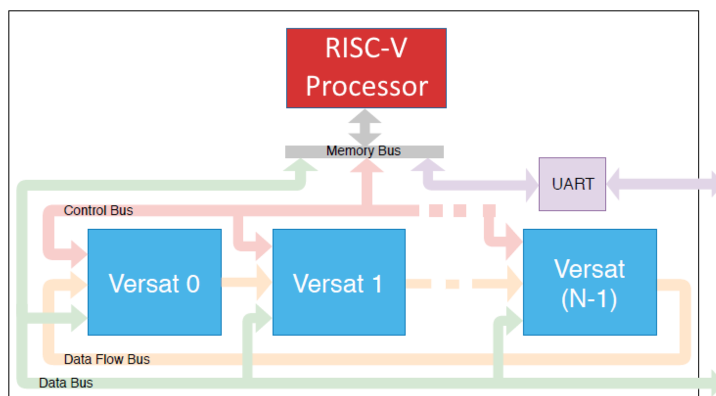


Figure 3.7: Deep Versat system, adapted from [13].

3.6.4 Deep Versat API

An API in C++ was created to facilitate datapath configuration in Deep Versat. Classes represent the hardware modules with their functions and connections being accessed by the class's methods.

First, the definition of the number of Versat layers, types and number of FUs and memory sizes is established. Then, the hardware architectural parameters of Deep Versat are passed into the API via a python script that generates a C++ header file from the Verilog header file containing all the macros used for pre-silicon configuration.

3.7 Conclusions

CGRAs have the potential to combine the computational capabilities of FPGAs with the versatility of software, lowering the barrier of entry for model deployment. For that, the CGRA requires dedicated FUs for the CNN layers that exploit datapath and model reduction optimizations, while having an adequate toolchain for software development. The Deep Versat CGRA provides scalability by connecting multiple Versat layers in a ring structure, while being controlled by a RISC-V CPU through a C++ API. This CGRA architecture will be used as basis for the newly developed CGRA architecture geared towards CNN acceleration.

Chapter 4

System Baseline

This chapter presents a performance baseline of the Tiny YOLOv3 application executed in the IOb-SoC system. Section 4.1 provides a brief overview of the adjustments that enable Tiny YOLOv3 execution on an embedded environment. Section 4.2 describes the IOb-SoC hardware platform used as the base system, including the peripherals used and their respective role in the system. The chapter concludes with a profiling of the target application in section 4.3.

4.1 Embedded Software Version

The execution of the Tiny YOLOv3 application in an embedded environment requires changing all memory allocations to static memory regions predefined at compile time due to lack of operating system. The data format is also changed from floating-point to 16-bit fixed point.

The CNN model execution also benefits from performance optimizations such as batch-normalize folding and linear approximations of the activation functions. The model accuracy is also improved by implementing dynamic fixed-point post-training quantization.

4.1.1 Batch Normalize Fusion

In the Tiny YOLOv3 neural network, batch normalization is applied to the MAC result of the convolution. The MAC result is given by (4.1).

$$y = b + \sum_{c=0}^{C-1}\sum_{j=0}^{J-1}\sum_{k=0}^{K-1} w_{cjk} \times x_{cjk} \tag{4.1}$$

Where b and w_{cjk} are the convolutional bias and weights associated with a specific convolution kernel. Using (4.1) as the input of (2.3) the batch normalization output is given by

$$z = Ay + B = A(b + \sum_{c=0}^{C-1}\sum_{j=0}^{J-1}\sum_{k=0}^{K-1} w_{cjk} \times x_{cjk}) + B = (Ab + B) + \sum_{c=0}^{C-1}\sum_{j=0}^{J-1}\sum_{k=0}^{K-1} (Aw_{cjk}) \times x_{cjk}, \tag{4.2}$$

which is equivalent to a convolutional MAC result with bias b' and weights w'_{cjk} given respectively by

$$b' = \frac{b - \mu}{\sqrt{\sigma^2 - \epsilon}}\gamma + \beta, \text{and } w'_{cjk} = \frac{w_{cjk}}{\sqrt{\sigma^2 - \epsilon}}\gamma. \quad (4.3)$$

This arithmetic manipulation reduces the number of inference parameters to use in the network, while maintaining the same accuracy. The new biases and weights are computed in floating-point from the original values. The biases and weights for the embedded neural network version are quantized to 16-bit values.

4.1.2 Activation Function Approximations

The leaky activation function described in figure 2.2(c) has $\alpha = 0.1$. To perform leaky activation while avoiding multiplication, α can be approximated by a sum of powers of 2. With this approach, the operation is replaced by a sum of multiple shifts. The α' value used to approximate the leaky activation is given by (4.4)

$$\alpha' = 2^{-4} + 2^{-5} + 2^{-7} = 0.1015625. \quad (4.4)$$

According to [37], a hardware implementation of a piecewise linear approximation for the sigmoid activation provides a tradeoff between resources used and reduced precision. Table 4.1 presents the linear approximations for each input range. Note that the input x is always multiplied by a power of 2 so that the multiplication in hardware can be performed by a shift.

Input Range $x \in$	Linear Approximation
$]-\inf; -5]$	$y = 0$
$]-5; -2.375]$	$y = 2^{-5}x + 0.15625$
$]-2.375; -1]$	$y = 2^{-3}x + 0.375$
$]-1; 1[$	$y = 2^{-2}x + 0.5$
$[1; 2.375]$	$y = 2^{-3}x + 0.0625$
$[2.375; 5]$	$y = 2^{-5}x + 0.84375$
$[5; +\inf]$	$y = 1$

Table 4.1: Linear sigmoid approximation.

4.1.3 Dynamic Fixed-Point

As discussed in section 3.4.1, using different fixed-point quantization for each layer can improve the model accuracy. The implemented dynamic fixed-point model allows for independent fixed-point formats for the activation, weights and biases in each layer. Table 4.2 presents the fixed-point format used to approximate the weights, biases and output values of each layer.

The weight and bias formats are determined from the original floating-point weight file. The number of integer bits is the minimum that accomodates for the detected value range. The output FM format is determined from the value range detected during the inference of the COCO [20] test dataset.

Only the Convolutional layers are referred since in the Maxpool, Upsample and Route layers the output has the same fixed-point format as the input. The Yolo layers are computed as a sigmoid activation after a convolution with linear activation. Both Yolo layers have a Q3.13 output FM format, the same as the preceding Convolutional layer.

Layer	Output FM	Weight	Bias	Layer	Output FM	Weight	Bias
1	Q8.8	Q6.10	Q5.11	14	Q6.10	Q1.15	Q2.14
3	Q7.9	Q1.15	Q4.12	15	Q6.10	Q1.15	Q3.13
5	Q7.9	Q2.14	Q5.11	16	Q3.13	Q2.14	Q3.13
7	Q7.9	Q1.15	Q4.12	19	Q6.10	Q2.14	Q3.13
9	Q7.9	Q1.15	Q4.12	22	Q6.10	Q1.15	Q3.13
11	Q7.9	Q1.15	Q4.12	23	Q3.13	Q1.15	Q4.12
13	Q8.8	Q2.14	Q4.12				

Table 4.2: Dynamic fixed-point quantization of output FMs, weights and biases of each layer.

4.1.4 Embedded Software Validation

The validation of changes to the neural network model requires the development of a new model for desktop execution accelerated by GPU. The GPU executes custom-developed kernels so that the complete execution uses integer arithmetic to emulate fixed-point operations. All the neural network models perform inference over the MS COCO 2017 test dataset [20]. Table 4.3 compares the obtained mAPs between the original neural network model and the embedded versions.

Tiny YOLOv3 Configuration	mAP_{50}
Floating-Point	32.9
16-bit FP + Leaky and Sigmoid approximation + Static Quantization	29.8
16-bit FP + Leaky and Sigmoid approximation + Dynamic Quantization	30.8

Table 4.3: mAP_{50} for several Tiny YOLOv3 network models.

The use of dynamic fixed-point quantization increases the mAP by 1 point. The final embedded model has a 2 point lower mAP when compared with the original model. This difference is related to the use of fixed-point instead of floating-point and the use of linear activation functions. Further mAP improvements require network re-training, for example using linear approximations for the activation functions. Since the re-training process for neural networks falls out of scope for the work, the model implemented uses DFP approximation.

4.2 IOb-SoC Hardware Platform

The baseline hardware system used for this work is IOb-SoC [10], an open-source RISC-V SoC platform developed by IObundle. Figure 4.1 presents the block diagram of the system. A RISC-V soft CPU core controls the system.

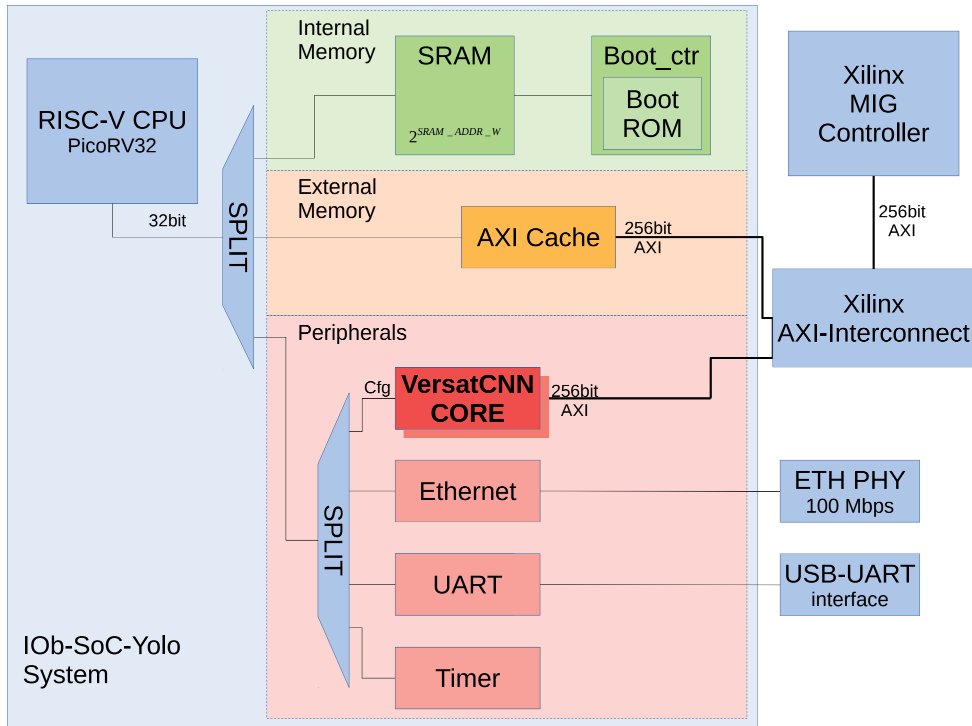


Figure 4.1: IOb-SoC-Yolo block diagram.

The CPU used is the PicoRV32 RISC-V core [38], minimally modified to be integrated into IOb-SoC. The CPU core implements the interger (I), atomic (A) and multiply/divide (M) RISC-V instruction set extentions and is programmed with the RISC-V GNU Compiler Toolchain [39]. The PicoRV32 CPU is designed to use minimal hardware resources and, as a consequence, has very low performance, taking 4 Cycles Per Instruction (CPI). The CPU can access the internal memory, the external DDR memory (via cache), and the four peripherals mentioned:

- the VersatCNN Accelerator Core is the main focus of this work and is used to accelerate the convolution operation.
- the Timer is used to measure performance.
- the UART is used for programming and basic user runtime messages.
- Ethernet module provides a higher bandwidth communication facility used to transfer large datasets to IOb-SoC.

The access to memory and peripherals is done through a native interface that follows a valid-ready protocol which performs one read or write transfer at a time. This interface uses six-ports, as represented in figure 4.2. The CPU, acting as master, initiates a transfer by asserting the *valid* output and holds until the *ready* signal is also asserted by the slave (either memory or peripheral). In the read transfer, the

slave reads the address (*addr*) and provides the read value (*rdata*) at the same clock cycle where the *ready* signal is asserted whilst the strobe (*wstrb*) and write data (*wdata*) signals are unused. In the write transfer, the slave writes the data to the address indicated and acknowledges the transfer by asserting the *ready* signal. The strobe is a 4-bit wide enable with one bit for each byte of the addressed word. In the write transfer case, the read data input is unused by the slave module.

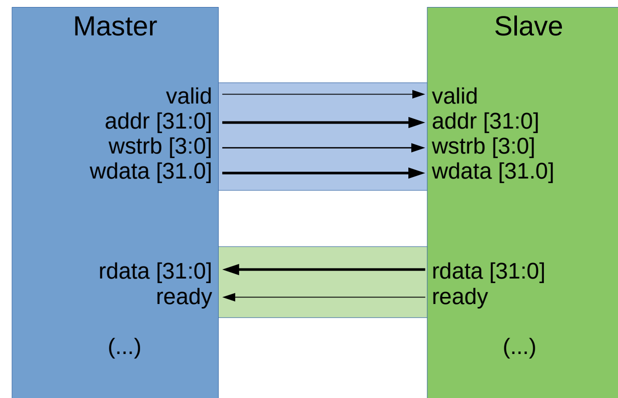


Figure 4.2: IOb-SoC native interface.

Table 4.4 shows the memory map to access the system memories and peripherals, where the address ranges reserved for each system component is indicated. The start addresses in each range are used as the software base address to access the component. Note that the address space of each component is generally lower than the reserved space.

Memory / Peripheral	Reserved Address Space
Internal Memory	0x0000 0000 - 0x1FFF FFFF
Boot Controller	0x2000 0000 - 0x3FFF FFFF
UART	0x4000 0000 - 0x4FFF FFFF
Timer	0x5000 0000 - 0x5FFF FFFF
Ethernet	0x6000 0000 - 0x6FFF FFFF
VersatCNN	0x7000 0000 - 0x7FFF FFFF
External / DDR Memory	0x8000 0000 - 0xFFFF FFFF

Table 4.4: IOb-SoC memory mapping.

For the Tiny YOLOv3 application, the CPU executes instructions from a program stored in the internal memory. Figure 4.3 presents the internal memory block diagram. The CPU can read from or write to the SRAM memory; additionally, the CPU can send a reset signal to the boot controller.

The boot controller contains a ROM memory with the bootloader instructions. After a hardware reset, the bootloader program is copied to the SRAM and runs from a specific address. The bootloader writes the firmware received from the UART to address 0 of the SRAM. When the reception and writing are complete, the bootloader soft resets the system and the CPU starts executing the received firmware.

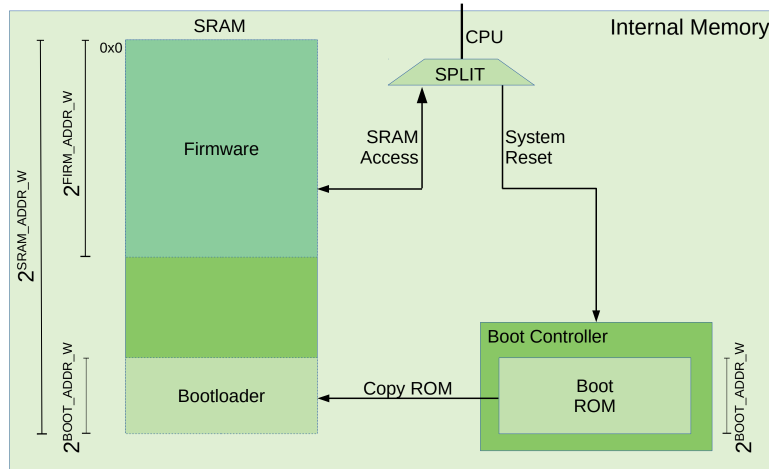


Figure 4.3: Internal memory block diagram.

This approach enables firmware updates without having to synthesize the hardware system for each change, as the new program can be loaded from the UART.

The bootloader program needs to fit inside the $2^{\texttt{BOOTROM_ADDR_W}}$ byte boot ROM module. The internal memory has a size of $2^{\texttt{SRAM_ADDR_W}}$ bytes. The internal memory is bounded by

$$2^{\texttt{SRAM_ADDR_W}} \geq 2^{\texttt{BOOTROM_ADDR_W}} + \texttt{FIRM_SIZE}, \tag{4.5}$$

where FIRM_SIZE is the byte size of the firmware program. The bootloader program has 3140 bytes, which requires that

$$\texttt{BOOTROM_ADDR_W} = \texttt{ceil}(\log_2(3140)) = 12. \tag{4.6}$$

Since the firmware has a size of 33796 bytes, from (4.5) follows that

$$\texttt{SRAM_ADDR_W} = \texttt{ceil}(log_2(3140 + 33796)) = 16. \tag{4.7}$$

Hence, the system requires 64kB of internal SRAM memory and 4kB of ROM to permanently store the bootloader program.

A single level cache developed by [12] accesses the external memory through an AXI4 interface. The cache can be parameterized with regards to word size, number of words per line and number of lines. Furthermore, the cache associativity (either directly-mapped or associative) and replacement policy (Least-Recently Used (LRU), PseudoLRU or tree-based PseudoLRU) are also configurable. The cache only supports write-through policy.

The cache also provices a C API to enable programmer-level control, such as the invalidation of the cached data, as well as performance metrics like read/write misses/hits or write buffer state.

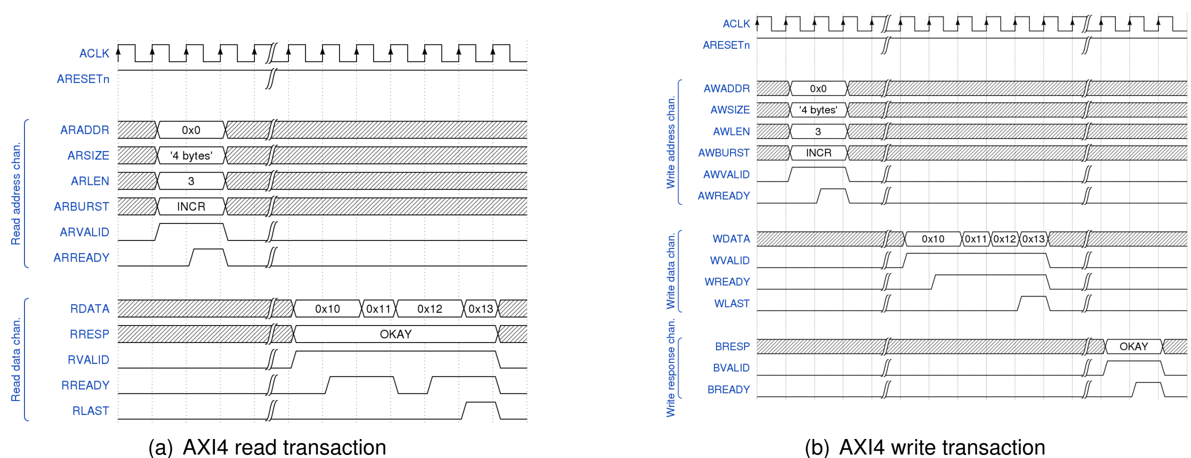
The AXI4 protocol supported by the cache is based on a valid/ready handshake. This protocol presents five separate channels: read address channel, read data channel, write address channel, write data channel and write response channel.

Figure 4.4(a) shows an example of a read transaction, which begins with the request from the master that asserts the address read valid (*ARVALID*) signal and provides the initial data address (*ARADDR*) along with control information to the read address channel. The control information mainly includes:

- **Burst size (*ARSIZE*)**: maximum number of bytes to transfer in each beat within a burst;
- **Burst length (*ARLEN*)**: number of transfers in a burst (maximum 256 transfer per burst);
- **Burst type (*ARBURST*)**: method for beat address calculation (FIXED, INCR or WRAP).

There are other auxiliary signals for specific transfers. After the read address handshake (slave asserts *ARREADY*), the slave responds with the requested data and status of each beat via the read data channel. Each beat is performed with a *RVALID/RREADY* handshake. In the last transfer beat, the slave asserts *RLAST* which informs the master that there are no more beats without a new read request.

Figure 4.4(b) shows an example of a write transaction, which starts with the request from the master providing the initial address from which data will be transmitted, among other information pretaining the burst configuration (*AWSIZE, AWLEN, AWBURST*) via the write address channel. After the *AWVALID/AWREADY* handshake, the master transfers the data in the write data channel, which ends when *WLAST* is asserted. To inform about the transfer status, the slave sends back to the master the status of the write transaction by setting the *BRESP* signal over the write response channel during the *BVALID/BREADY* handshake.



(a) AXI4 read transaction

(b) AXI4 write transaction

Figure 4.4: Example of an AXI4 read and write transaction (adapted from [40])

The cache for the baseline performance profiling uses the default configurations comprised of associative cache with 4 ways, 16 lines per way and 16 words of 32 bits per line. The default configuration also uses the LRU replacement policy. The cache size for the default configuration is 4kB.

For the final system with the VersatCNN accelerator, the cache is directly mapped (non-associate), uses the write-through/no allocate write policy and the LRU line replacement policy. The cache size is 1kB: 16 lines of 16 words of 32-bit each.

The reduction in cache size is justified by the use case for the cache in the final application. Since the VersatCNN accelerator has a DMA, the CPU only accesses the main memory directly to initially write the weights and input image and read the output and final image with the detected objects in the end. These operations consist of large transfers of sequential data. Therefore, adding more ways, lines or increasing the line width of the cache is useless and cannot improve performance.

4.2.1 Peripherals

As outlined previously, the CPU controls multiple peripherals. This section describes the general-purpose peripherals UART, Timer and Ethernet and contextualizes their use in the final application. Chapter 5 describes in detail the VersatCNN CGRA, used to accelerate the Tiny YOLOv3 application.

4.2.1.1 UART

The UART peripheral is already included by default in the IOb-SoC platform and is used to transfer the application firmware during the bootloader execution and to print debugging information on the host machine connected to the FPGA board via USB. The UART allows for asynchronous reads and writes using full-duplex transmit and receive channels, where each channel uses a single serial wire. Fig. 4.5 explains the UART data frame.

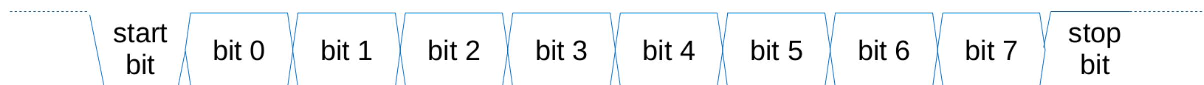


Figure 4.5: UART data frame.

The line is set to high by the transmitter as default while it is idle. When a byte is transmitted, the signal is set to low followed by the 8 bits of the byte. After sending the byte, the line returns to high signalling the end of the transmission. The data rate for the UART is defined by the baud rate which is the number of bits transferred per second. The baud rate is related to the system clock (f_{clk}) by

$$\texttt{Baud Rate} = \left\lfloor \frac{f_{clk}}{\texttt{DIV}} \right\rfloor, \tag{4.8}$$

where `DIV` is the integer division quotient of the system clock frequency over the baud rate. The baud rate used for UART communications is 115200. This baud rate represents a transfer rate of 14.4kB/s.

Figure 4.6 presents the communication protocol implemented between the console application on the PC and the embedded system. The connection process starts with the embedded system, executing the bootloader program, sending an enquiry (*ENQ*) byte in a loop until the console answers with an acknowledge (*ACK*) byte. After that, the console sends the firmware binary size, followed by the firmware

binary, which is saved in the SRAM. With the firmware sent to the embedded system, the console sends a *RUN* command that triggers the soft reset process and starts the execution of the firmware program. During the firmware program, the console only prints the bytes received from the UART connection. At the end of the firmware program, the embedded system sends an End of Transmission (*EOT*) byte to the console that closes the connection.

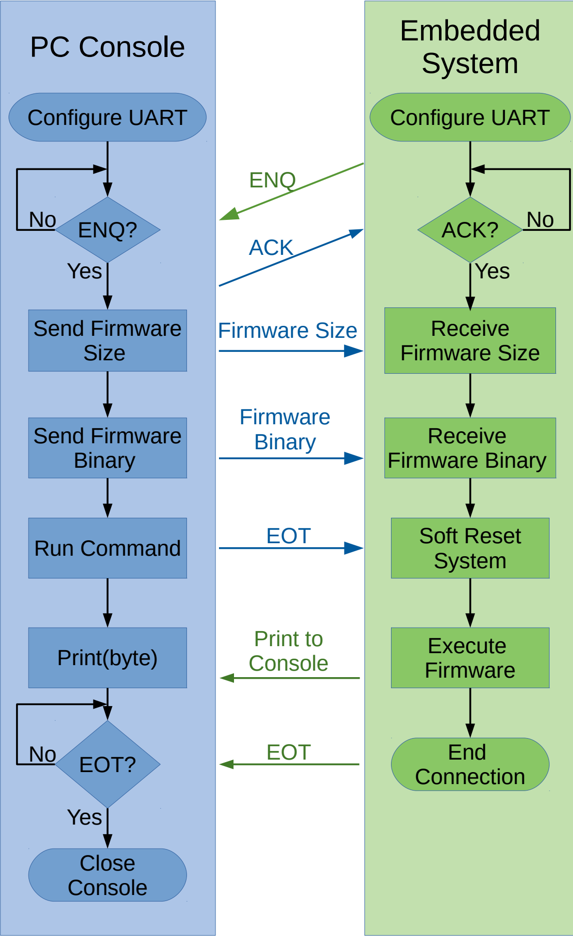


Figure 4.6: UART communication protocol.

4.2.1.2 Timer

The timer module is a 64-bit counter that increments every clock cycle. The CPU can reset this counter or save its value to a register at a particular time.

The timer module has C driver functions that return the elapsed time with a specific resolution which dictates the timer period. Tab. 4.5 presents the resolutions and the corresponding periods. Naturally, the timer period is smaller for finer timer resolutions.

Timer Resolution	Timer Period
1 μs	1h11min
1 ms	49 days
1 s	136 years

Table 4.5: Timer resolutions and corresponding periods.

4.2.1.3 Ethernet

The ethernet module is used in the system to initially transfer the neuron's weights, input image and output image. These files have a size of multiple MB, so using the UART module for transferring them would take too long.

The ethernet module uses raw sockets for data transfers. Fig. 4.7 presents the structure of the ethernet data frames.

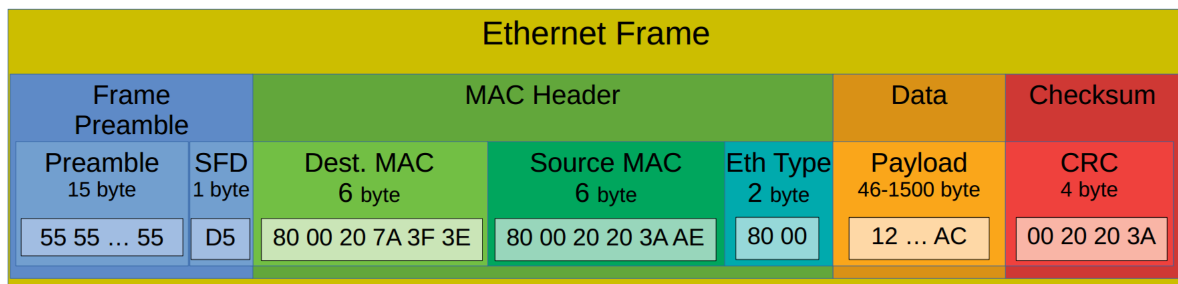


Figure 4.7: UART data frame.

The data frame starts with a 15-byte preamble composed of alternating zeros and ones. The preamble is terminated by a start frame delimiter (SFD) byte with value 0xD5.

The destination and source Medium Access Control (MAC) addresses follow the SFD. For frames received by the system, the destination MAC address is the Ethernet module MAC address, and the source MAC address is that of the host machine that communicates with the FPGA.

After the MAC addresses, 2 bytes indicate the frame protocol. In this case, the protocol used is the IPv4 datagram, so the EtherType value is 0x8000.

The frame payload is sent after the MAC header. The payload can have from 46 up to 1500 bytes. Both sender and receiver need to be configured to use the same frame sizes. The final field in the frame is the CRC checksum, used to detect transmission errors. For the developed system, the Ethernet frames have 1006 bytes of payload.

By using the Ethernet C driver functions, the construction of Ethernet frames for sending or the extraction of payloads when receiving is transparent from a firmware perspective. The firmware developed initializes the Ethernet drivers and sets the Ethernet frame size.

The host machine uses a python script for Ethernet communications. It sends the initial image and weights and receives the final image.

4.3 Tiny YOLOv3 Profiling

Table 4.6 presents the execution times for the Tiny YOLOv3 application on the presented IOb-SoC with a frequency of 143MHz. The compiled program uses `-O3` optimizations. As expected, the CNN is resposible for the majority of the computation, taking 99.9% of the total execution time.

The profiling also includes the CNN pre and post-processing functions of the application. The CNN pre-processing resizes the input image from size $Img_W \times Img_H \times Img_C$ to size $416 \times 416 \times 3$, which is the required size for processing. The CNN post-processing searches the CNN output for scores over a threshold value, applies NMS to the potential detections and draws the detections into the input image. In addition to the CNN, both processes require acceleration to achieve the target performance.

Tiny YOLOv3 Part	Execution Time (s)
Pre-CNN	1.040
CNN	967.746
Post-CNN	0.014
Total	968.800

Table 4.6: Tiny YOLOv3 RISC-V-only performance.

Table 4.7 presents the execution time per CNN layer type. The Convolutional layers are responsible for 99.8% of the CNN execution time. Both the Maxpool and Yolo layers have an execution time over the 33.3ms target performance. The Upsample layers take about 50% of the target execution time. With these profiling results follows that all the layer types require acceleration to achieve the target performance of 30 FPS.

CNN Layer Type	Execution Time (s)
Convolutional	966.026
Maxpool	1.546
Upsample	0.017
Yolo	0.157
Total	967.746

Table 4.7: CNN execution time per layer type.

4.4 Conclusions

The software modifications and optimizations enable the implementation of the Tiny YOLOv3 application in an embedded environment, while minimizing the impact in the model accuracy. The IOb-SoC platform provides a development system controlled by a RISC-V CPU. Software using specific C drivers controls the peripherals. The use of a bootloader program that loads the firmware via UART permits changes to the firmware without re-synthesizing the hardware. The UART data transfer rate is too low for the amount of data required by the application. Thus, an Ethernet module transfers input images, weights and

output images. The timer module measures times, profile the application and assess the accelerations achieved. The profiling results of the embedded software executed on the IOb-SoC platform show that all the CNN requires acceleration accross all layer types. Additionally, the CNN pre and post-processing parts of the application also need acceleration to achieve the target performance of 30FPS.

Chapter 5

VersatCNN CGRA

This chapter describes the main features of the VersatCNN CGRA hardware architecture developed in [9]. Section 5.1 introduces the overall CGRA architecture, highlighting the improvements done to the original Deep Versat core. Section 5.2 introduces the address generator unit, responsible for generating the data access patterns for the memories in the core. Section 5.3 details the developed custom functional unit used for computations and explores the relevant configurations to accelerate the different neural network layers. The chapter concludes with a brief overview of the CGRA API in section 5.4.

5.1 Architecture

The versatCNN is the result of multiple architectural improvements to the Deep Versat CGRA developed in [13]. Figure 5.1 presents the detailed architecture of the VersatCNN Core.

The VersatCNN uses an AXI DMA (Direct Memory Access) to read and write data from the main memory. Bypassing the CPU for the memory accesses frees the CPU to execute other tasks during data transfers to/from the VersatCNN. The use of DMA also increases the memory bandwidth of the core. The DMA has 256-bit data width and with AXI bursts the bandwidth averages to almost one 256-bit word per cycle for both reads and writes.

The data from main memory is stored in buffer memories called vReads. There are two sets of vRead FUs in the architecture: one for IFMs and another for weight kernels and biases. The vRead FUs send the data into the custom computational FUs.

The custom FUs are organized in a two-dimensional matrix to enable parallelism for loop 4 and loop 3, as discussed in section 3.3. Loop 4 uses the same input feature map values and different kernels. Loop 3 uses the same kernels and different input feature maps.

The results from each custom FU line are concatenated and written into an output memory buffers called vWrite. The data from the vWrite is transfered back into main memory through the DMA.

The VersatCNN dataflow is divided into three phases: *memory read*, *compute* and *memory write*, as is presented in figure 5.2.

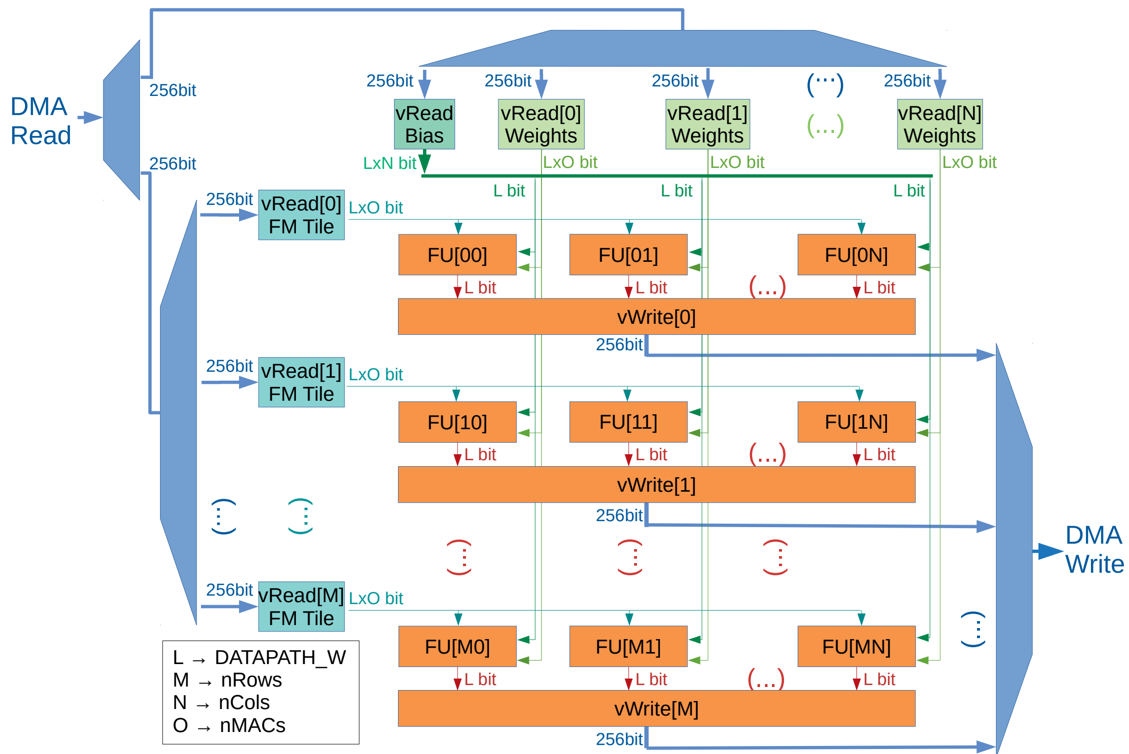


Figure 5.1: VersatCNN detailed block diagram.

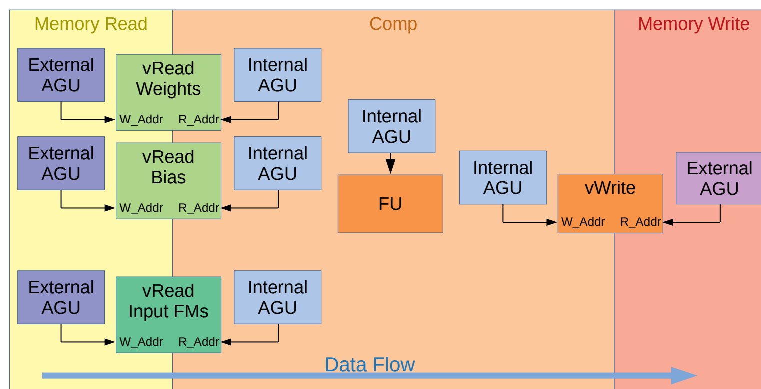


Figure 5.2: Address generator distribution diagram.

In the *memory read* phase, the data is transferred from the main memory to the vRead FUs. In this phase, the read operations from the main memory and the writes to the vRead FUs are controlled external address generator units (AGUs).

The *compute* phase sends the data from the vRead FUs for computation in the custom FUs and writes the result in the vWrite FUs. In this phase, all the internal AGUs control the reads from the vRead FUs, the operations at the custom FUs and the writes to the vWrite FUs.

In the *memory write* part the data from the vWrite FUs is transferred back to main memory. A external AGU controls the read accesses to the vWrite FUs and the write transfers to main memory.

This architecture allows the pipelining of consecutive dataflows. The execution of a second dataflow starts after the *memory read* phase of the first dataflow is completed. At that point, the CGRA executes the *memory read* phase of the second dataflow and the *compute* phase of the first dataflow simultaneously.

Table 5.1 presents the VersatCNN parameters.

Parameter	**Description**
nCols	Inter-FM parallelism factor
nRows	Intra-FM parallelism factor
nMACs	Inter-convolution parallelism factor
DDR_ADDR_W	Main memory address width
DATAPATH_W	Computation data width
VREAD_BIAS_ADDR_W	Bias memory size
VREAD_WEIGHT_ADDR_W	Weight memory size
VREAD_TILE_ADDR_W	Input FM memory size
VREAD_PATTERN_ADDR_W	External address pattern memory size
VREAD_TILE_EXT_ADDR_W	External AGU port dimentions
VWRITE_ADDR_W	vWrite memory size

Table 5.1: VersatCNN parameters.

The number of vRead FUs for kernels used in the design is defined by the nCols parameter. The number of vRead FUs used for the input feature maps is defined by the nRows parameter. With this design, the number of vRead FUs varies with $\texttt{nCols} + \texttt{nRows}$ and the number of vWrite FUs with nRows, while the number of custom FUs for computation is $\texttt{nCols} \times \texttt{nRows}$.

The nMACs parameter defines the number of MACs in each custom FU.

The DDR_ADDR_W parameter sets the main memory address width.The DATAPATH_W parameter determines the data width used for computation input and output.

The VREAD_TILE_EXT_ADDR_W parameter defines the port dimensions of the external AGUs for the vRead memories that store the input FMs.

The remaining parameters VREAD_BIAS_ADDR_W, VREAD_WEIGHT_ADDR_W, VREAD_TILE_ADDR_W, VWRITE_ADDR_W define the dimensions for the respective bias, weight, IFM and vWrite memories.

The VREAD_PATTERN_ADDR_W parameter defines the dimensions of a memory that stores an external address pattern.

5.2 Address Generator

The vRead and vWrite FUs, as well as the execution of the custom FUs, are controlled by AGUs. Figure 5.2 displays the distribution of the AGUs across the VersatCNN. The versatCNN uses a total of

nine groups of AGUs. The buffer memories are divided into four groups: vReads for weights, a single vRead for biases, vReads for input FMs and vWrites for OFMs. Each group of memories is controlled by a pair of AGU groups: one for the external accesses and another for the internal accesses. All the custom FUs for computation use a single AGU to generate control signals.

Figure 5.3 presents the inputs and outputs of the address generator unit.

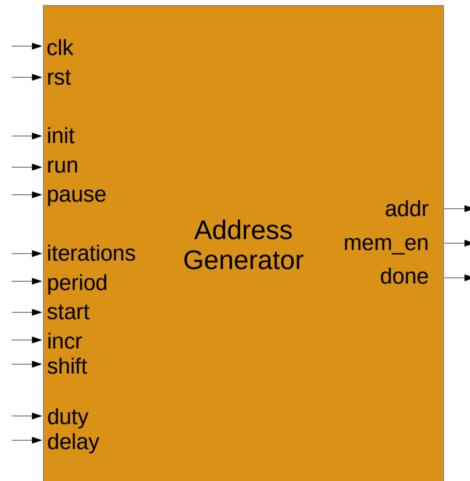


Figure 5.3: Address generator ports.

The `init` signal initializes the address generator with the starting values and the `run` signal starts the address generation process. The VersatCNN hardware controls these signals.

The inputs `start`, `iterations`, `period`, `incr` and `shift` determine the pattern outputted in the `addr` port. An address generator unit can access the memories in a nested loop pattern, which is described by algorithm 2.

addr = start
for $i \in \{1, \ldots, iterations\}$ **do** // Outer loop
  for $p \in \{1, \ldots, period\}$ **do** // Inner loop
    addr += incr
  addr += shift

Algorithm 2: Address generator pattern.

The `duty` sets the number of `period` cycles that the address is incremented. So if $\texttt{period} = 3$ and $\texttt{duty} = 2$, the address is only incremented 2 of the 3 period cycles. The `delay` sets an initial latency of clock cycles between enabling `run` and actually starting the address generation.

The `start`, `iterations`, `period`, `incr`, `shift`, `duty` and `delay` signals are configured via software.

The `addr` corresponds to the address generated and the `mem_en` to the enable signal for the access. The `done` signals the completion of the address pattern.

A pattern with more nested loops is achieved by chaining multiple AGUs. Each additional chained address generator unit adds a set of `iter`, `period`, `shift` and `incr` configurations.

The AGU for the internal access of the IFMs vReads uses three chained simple AGUs since the access pattern required for the convolution has six nested loops.

The external address generation also requires the `offset` and `pingpong` configurations. The `offset` is the address space between each starting address to read or write from/to main memory. The `pingpong` enables the buffers to work as two independent halves: one half is used in a phase to read and the other to write. They switch in the next phase execution.

5.3 Custom Functional Unit

The custom FU for computation receives as input the feature map values, biases and weights from the vRead FUs. The custom FU outputs the computation result to a vWrite FU.

The custom FU for computation enables the parallelism of loop 2, where multiple IFM channels are computed in parallel using multiple MACs, as discussed in section 3.3. This parallelism requires multiple words of IFM data and kernel weights.

The number of input feature map channels computed in parallel is parameterizable.

The custom FU function is controlled by the `bypass_adder`, `maxpool` and `bypass` signals. Table 5.2 presents the main functions performed by the custom FU and their respective configuration mapping. These functions enable the acceleration of different layer types.

Custom FU Function	bypass_adder	maxpool	bypass
MAC output	1	x	x
Maxpool	0	1	1
Convolution and Maxpool	0	1	0
FM_Data	0	0	1
Convolution	0	0	0

Table 5.2: Custom FU main functions

The MAC output function shifts the output of a single MAC block. This function enables simple MAC operations for multiple quantization formats.

The convolution function takes the outputs of the several MAC blocks and adds them together. The result of the convolution can be used as-is or be subject to either leaky or sigmoid non-linear activation. The non-linear activations are implemented for fixed-point as discussed in section 4.1.2. The FU shifts the final result according to the quantization format being used.

For the convolution and maxpool function, each computed convolutional result is compared to the previous convolution results of the same 2×2 maxpool region.

The standalone maxpool function performs the same logic but using directly one of the words from the IFM data.

The FM_Data function bypasses both the convolution and maxpool logic, outputting one of the input feature map data words.

The configurations for the convolutional operations are set by the `leaky`, `sigmoid` and `bias` signals. All valid configurations are presented in table 5.3.

Convolution Type	leaky	sigmoid	bias
Bias and Leaky activation	1	x	1
Leaky activation	1	x	0
Bias and Sigmoid activation	0	1	1
Sigmoid activation	0	1	0
Bias and no activation	0	0	1
No activation	0	0	1

Table 5.3: Custom FU convolution functions and configurations.

The Tiny YOLOv3 layers presented in table 2.3 can be accelerated in the VersatCNN using the custom FU configurations presented in table 5.4.

Tiny YOLOv3 Layers	bypass_adder	maxpool	bypass	leaky	sigmoid	bias
(1-2);(3-4);(5-6);(7-8)	0	1	0	1	x	1
9;11;13;14;15;19;20;22	0	0	0	1	x	1
10;12	0	1	1	x	x	x
(16-17);(23-24)	0	0	0	0	1	1

Table 5.4: Custom FU configurations for Tiny YOLOv3 layers.

The layers 1 to 8 can be computed in Convolutional + Maxpool pairs. These layers disable the convolution bypassing and activate the maxpooling. All Convolutional layers have leaky activation.

The layers 10 and 12 are Maxpool only. These layers bypass the convolution, ignoring all convolution specific configurations.

The (16-17) and (23-24) layers are Convolutional + Yolo pairs. These layers can be computed together since the convolution has linear activation and the Yolo layer consists in performing sigmoid activation to the input feature maps. So these two layer types can be combined into a convolution with sigmoid activation. Note that only part of the Yolo layer output channels is activated. The VersatCNN has a mask configuration that controls the usage of the `sigmoid` signal for each custom FU column. The remaining layers perform convolution with leaky activation. All Convolutional layers in the neural network use bias.

The custom computational FU is controlled by an AGU. The AGU period controls the number of operations required to obtain one result. For example, a maxpool requires period $2 \times 2 = 4$, while a convolution with a $3 \times 3 \times 16$ kernel takes $3 \times 3 \times 16/\texttt{nMAC}$ period. Note the division since `nMAC` input channels are computed in parallel. The number of final outputs for a phase execution is defined by the iterations of the address generator.

5.4 API

The VersatCNN has a C++ API that enables runtime configuration by a soft CPU core. Figure 5.4 presents the class diagram of the API.

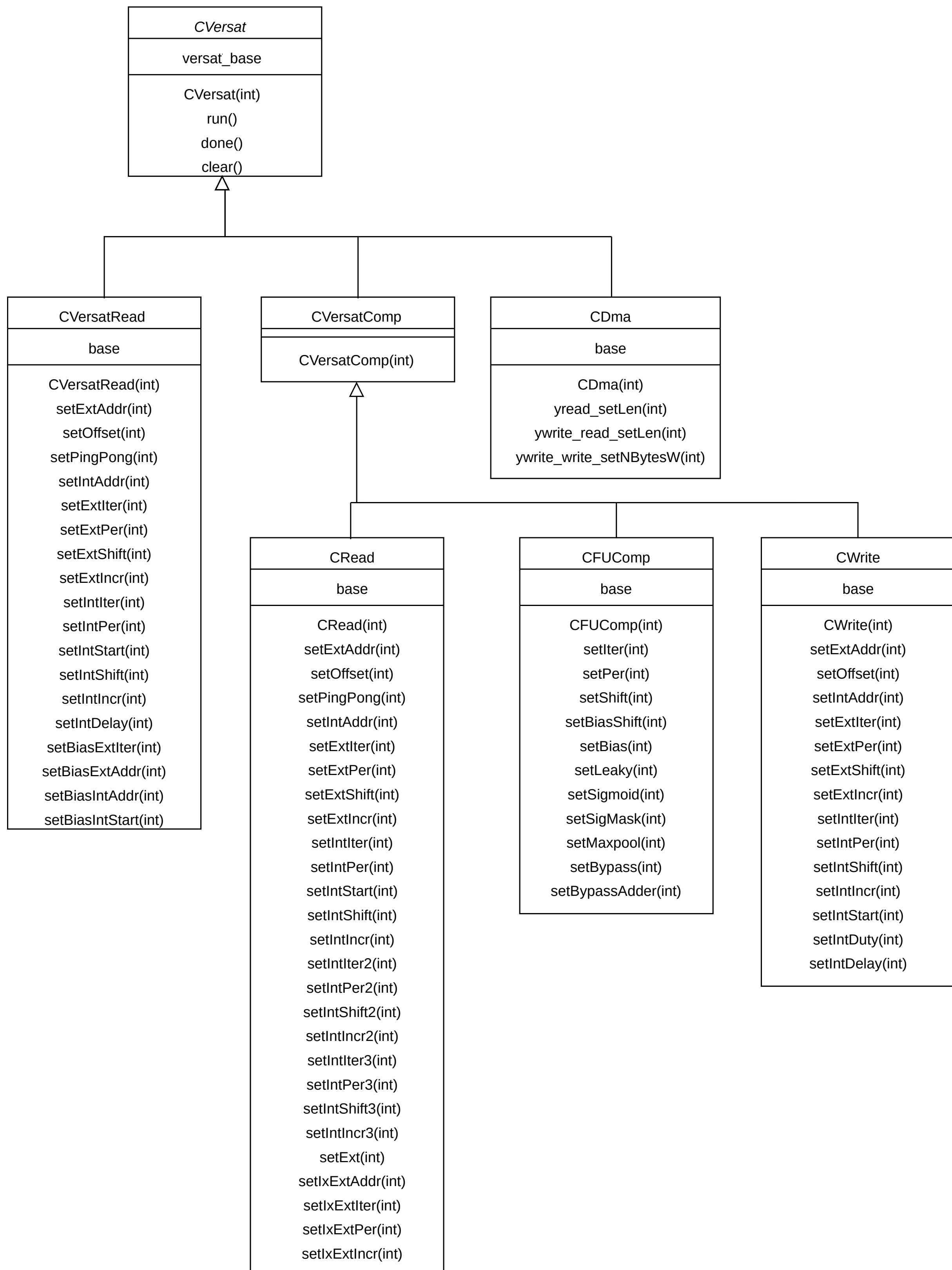


Figure 5.4: Class diagram of the VersatCNN CGRA.

The main class `CVersat` has an attribute for the memory base address of the peripheral in the system. Besides the constructor, the class has the `run()`, `done()` and `clear()` methods. The `run()` method

signals the execution of a dataflow phase in VersatCNN. The `done()` checks for a `run()` completion. The `clear()` method resets all VersatCNN configurations.

The `CVersat` class contains three objects, each from a different class. Each class contains methods to configure specific FUs in the CGRA. The `CDma` class has methods for the DMA configurations. The `CVersatRead` class has methods for the AGUs associated with the vReads for the weights and biases. The `CVersatComp` class contains objects from classes that in turn control the AGUs associated with the IFM vReads, custom computational FUs and vWrites. The custom computational FU configurations are also configured with methods from this class.

5.5 Conclusions

This chapter explains the architecture of the VersatCNN CGRA. The VersatCNN enables parallel computation of different OFMs, pixels from the same OFM and different input data channels. The custom computational FU enables acceleration of convolutional, maxpool, simple multiplication and bypass operations. The different forms of parallelism allow a mapping of the Tiny YOLOv3 network layers that guarantees the use of all computational FUs in every run. Most of the VersatCNN configurations done in software set the patterns of the AGUs that control the accesses to the memory buffers and part of the custom FU. The chapter closes with a brief overview of the VersatCNN API.

Chapter 6

Tiny YOLOv3 with VersatCNN

This chapter details the development and implementation of the Tiny YOLOv3 application to execute efficiently on the VersatCNN. Section 6.1 presents the multiple parts that comprise the complete application. Section 6.3 explains in detail the different configurations used for the CNN execution in the CGRA. Section 6.4 presents the CNN pre-processing algorithm and the VersatCNN configurations done to accelerate the algorithm execution. Section 6.5 overviews the CNN post-processing functions and presents the developed configuration that accelerates the execution of this part of the application.

6.1 Application Overview

Figure 6.1 presents the multiple parts that compose the complete Tiny YOLOv3 application developed in this work.

The application receives an input image of dimentions $\mathtt{IMG_W} \times \mathtt{IMG_H}$. The input image is resized to 416×416 resolution in the CNN pre-processing part. The resized image is the CNN input. The results from the CNN are post-processed in three main steps. The application checks for detections with objectness score over the pre-defined threshold. The next step filters overlapping boxes using NMS, as discussed in section 2.5. The filtered boxes and respective classes are drawn in the original input image.

The final software version developed has over 2200 lines of C code for a RISC-V platform in constant development. Listing 6.1 presents the pseudo-code of the top-level embedded software.

The profiling done in section 4.3 shows that all parts of the Tiny YOLOv3 application require acceleration to meet the real-time (30 FPS) objective. The final program for the application also includes functions for initial setup and system configuration. These functions include initial peripheral configuration, resetting the main memory and the transfer of the weight file and input image into the main memory. The initial setup also includes functions to calculate auxiliar arrays for pre and post-CNN processing.

The application has two functions for CNN pre-processing, which configure VersatCNN dataflows. The CNN inference is done by a sequence of different configuration strategies that accelerate distinct layer types and combinations. Note that the route layers are inferred by pointer manitulation. The post-

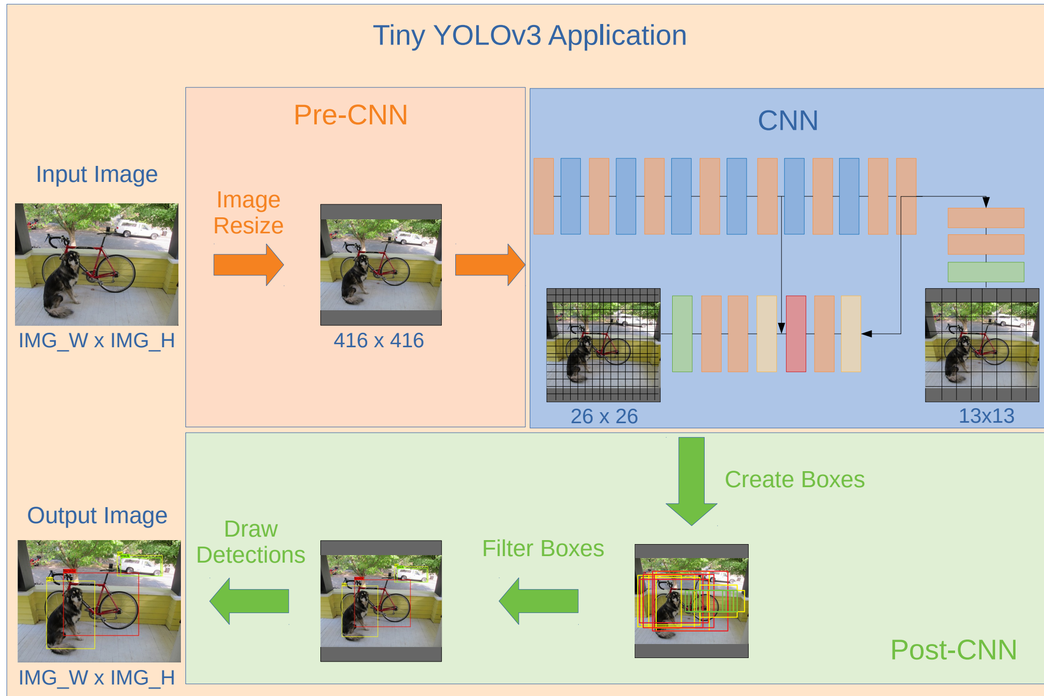


Figure 6.1: Tiny YOLOv3 application diagram.

CNN processing is divided in four functions, only the last function constitutes a VersatCNN configuration, the remaining are executed on the CPU.

The embedded software concludes by printing the detected objects to console and sending the output image to the PC via ethernet. The console program running on the PC is terminated at the end of the software.

The execution time is counted from before the CNN pre-processing until the detections are drawn on the output image. The peripheral initializations, main memory resetting and weight file transfer are excluded from the performance analysis because these routines are only performed once when the system is initialized. The image transfer is also excluded from the performance analysis since establishing a real-time image feed falls outside of the scope of the work.

6.2 Data Storage Format

The data storage format implemented in the embedded software influences the loops for convolution. To reduce the number of loops for data access during convolution, the data is stored in memory in a ZXY format instead of the original XYZ format. Figure 6.2 presents a scheme of both formats for a $4 \times 4 \times 3$ input feature map.

The XYZ format stores the values by column and row of each feature map. In ZXY format, the data is stored by channel first. Figure 6.2 also highlights the values used to compute the first pixel of a convolution output. The XYZ format disperses the input values in 9 groups of 3 contiguous values. This

```c
void Tiny_YOLOv3() {
  //initialize peripherals
  init_peripherals(); //UART, Timer, Ethernet, VersatCNN

  //Initial setup
  reset_DDR();
  rcv_data(); // receive input files from PC
  prepare_resize(); //calculate pre-CNN auxiliar arrays
  compute_RGB(); //compute RGB values for detections

////// APPLICATION START
  start_timer(); //reset timer counter
  //Pre-CNN
    width_resize();
    height_resize();
  //CNN
    convolution_and_maxpool(layer1_parameters); // layer 1+2
    convolution_and_maxpool(layer3_parameters); // layer 3+4
    convolution_and_maxpool(layer5_parameters); // layer 5+6
    convolution_and_maxpool(layer7_parameters); // layer 7+8
    convolution(layer9_parameters); // layer 9
    maxpool(layer10_parameters); //layer 10
    convolution(layer11_parameters); // layer 11
    maxpool(layer12_parameters); //layer 12
    convolution(layer13_parameters); // layer 13
    convolution(layer14_parameters); // layer 14
    convolution(layer15_parameters); // layer 15
    convolution_and_yolo(layer16_parameters); // layer 16+17
    // route layer 18
    convolution_and_upsample(layer19_parameters); // layer 19+20
    // route layer 21
    convolution(layer22_parameters); // layer 22
    convolution_and_yolo(layer16_parameters); // layer 23+24
  //Post-CNN
    create_boxes(yolo_layer_1); //create boxes from 1st yolo layer
    create_boxes(yolo_layer_2); //create boxes from 2nd yolo layer
    filter_boxes();
    draw_detections();
  stop_timer(); //stop timer counter
////// APPLICATION END
  print_results(); //detected objects and probability scores
  send_output_img(); //send output image to PC
  close_console();//terminate PC console connection
}
```

Listing 6.1: Embedded Software Program

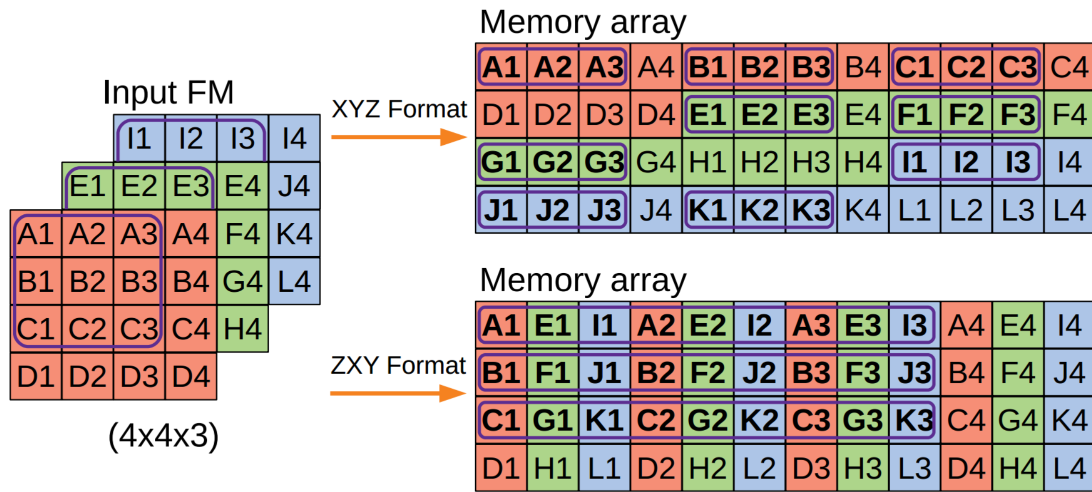


Figure 6.2: Data storage formats.

format requires 3 loops to iterate across all values: one loop to iterate across the values in each FM row, one loop to iterate across the rows of an FM and one to iterate across different FMs. The ZXY format stores the input values in 3 groups of 9 contiguous values each. This format requires 2 loops to access all values: one loop to iterate across all rows and one loop to iterate, in each row, across all values and channels.

6.3 CNN Acceleration

As discussed in section 4.3, all types of layers require acceleration. This section presents the different strategies implemented for acceleration.

6.3.1 Convolution with Maxpool

The first 8 layers of the network can be executed in Convolutional+Maxpool layer pairs. To minimize data transfers, each IFM tile is read from main memory once and the corresponding OFM tile is computed for all kernels. With this strategy, all the kernels are loaded from memory for the same set of IFM tiles. For the first four Convolutional layers, the kernels are small enough to fit into the vRead memory buffers at the same time, therefore the kernels are only loaded once from main memory. Listing 6.2 presents the code for the strategy implementation.

Most configurations stay the same across the complete Convolutional and Maxpool layers execution. The common configurations are set at the start of the function. Note that all the dataflow phases are configured.

The variable configurations are mostly related to updating the data pointers for reading and writing data to/from main memory.

The configurations for each FU type and dataflow stage are presented in isolation to highlight all configured data patterns for the Convolutional and Maxpool layer acceleration.

```
void Convolution_and_Maxpool(){

    update_ptrs(w_pos, ifm_pos, ofm_pos);

  // FIXED CONFIGURATIONS

    // Memory Read
    weights_bias_Ext_AGU_cfg();
    IFM_Ext_AGU_cfg();

    // Compute
    weights_bias_Int_AGU_cfg();
    IFM_Int_AGU_cfg();
    customFU_cfg();
    vWrite_Int_AGU_cfg();

    // Memory Write
    vWrite_Ext_AGU_cfg();

  // VARIABLE CONFIGURATIONS

    for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles

      for(j=0;j<W/Tile_W;j++){ // for each input tile in a row
        read_IFM_DDR(j, k); // read set of IFM tiles

        for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
          weights_bias_variable_cfg();
          vWrite_variable_cfg(j, k, l);

          while(versat.done()==0); // wait for dataflow part completion
          versat.run(); // execute new dataflow part

          stop_IFM_DDR_reads();
        }
        stop_weights_bias_DDR_reads();
      }
    }
    versat.clear(); // clear all configurations
  }
```

Listing 6.2: Convolutional+Maxpool acceleration

```
// Fixed configurations
versat.dma.yread_setLen(ker_side*ker_side*C);
versat.versatRead.setOffset(2*(ker_side*ker_side*C));
versat.versatRead.setExtPer((ker_side*ker_side*C)/16);
versat.versatRead.setExtIncr(16);
versat.versatRead.setExtIter(1);
versat.versatRead.setBiasExtIter(1);

// Variable configurations
for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles
  for(j=0;j<W/Tile_W;j++){ // for each input tile in a row
    for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
      versat.yread.setExtAddr(weights_ptr + 2*(nCols + l*nCols*(1+
          ker_side*ker_side*C)));
      versat.yread.setIntAddr(((ker_side*ker_side*C)*l)/16);
      versat.yread.setBiasExtAddr(weights_ptr + 2*l*nCols*(1+ker_side*
          ker_side*C));
      versat.yread.setBiasIntAddr(l);

      //versat.run();
    }
    versat.yread.setExtIter(0);
    versat.yread.setBiasExtIter(0);
  }
}
```

Listing 6.3: Weights and bias configurations for the *Memory Read* phase

6.3.1.1 Memory Read Phase Configurations

Listing 6.3 details the VersatCNN configuration to transfer a set of `nCols` kernels and biases from main memory. The set of kernels and biases are transferred in the same DMA burst, as they are stored sequentially in main memory.

The `yread_setLen()` method configures the burst length of the DMA transfer. Each kernel and corresponding bias amount to $\texttt{ker_side} \times \texttt{ker_side} \times \texttt{C} + 1$ values. Each transfer reads 16 values from memory (each value is 16-bit and $256 = 16 \times 16$). The burst length is obtained by dividing the total data to be transfered by the data per burst beat. The burst length is subtracted by 1, since the AXI4 specification [40] determines that

$$\texttt{Burst_Length} = \texttt{AxLEN} + 1. \quad (6.1)$$

The configurations in lines 3-6 control the pattern to write to the vReads. The `setOffset` indicates the byte offset between each kernel. The `setExtPer()` determines the number of operations per vRead, given by the kernel size divided by the number of values per transfer. Each operation writes 16 values into the vRead memory. After each write operation, the address is incremented by 16 (line 5). Lines 6 and 7 enable the writing process.

```
// Fixed configurations
versat.dma.ywrite_read_setLen((C*(Tile_W+2))/16-1);
versat.versatComp.read.setOffset(2*(2*((W+2)*C)));
versat.versatComp.read.setExtPer((C*(Tile_W+2))/16);
versat.versatComp.read.setExtIncr(16);
versat.versatComp.read.setExtShift(((W+2)*C) - (C*(Tile_W+2)));
versat.versatComp.read.setPingPong(1);

// Variable configurations
for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles
  for(j=0;j<W/Tile_W;j++){ // for each input tile in a row
    versat.ywrite.read.setExtIter(ker_side+1);
    versat.ywrite.read.setExtAddr(IFM_ptr+2*(k*2*((W+2)*C)*nRows+j*
        Tile_W*C));
    for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
      //versat.run()
      versat.ywrite.read.setExtIter(0);
    }
  }
}
```

Listing 6.4: IFM tile configurations for the *Memory Read* phase

The pointers to main memory and to the vReads for the kernels and biases are updated for each set read from memory. Lines 16 and 17 disable weights and biases reads from main memory since all kernels and biases are loaded into the vRead buffers after the first set of IFM tiles is processed.

Listing 6.4 presents the configuration to read IFM tiles from main memory.

The limitations of the on-chip memory buffers require the use of tiling for the IFMs. Figure 6.3 presents a diagram of the tiling strategy used for the application.

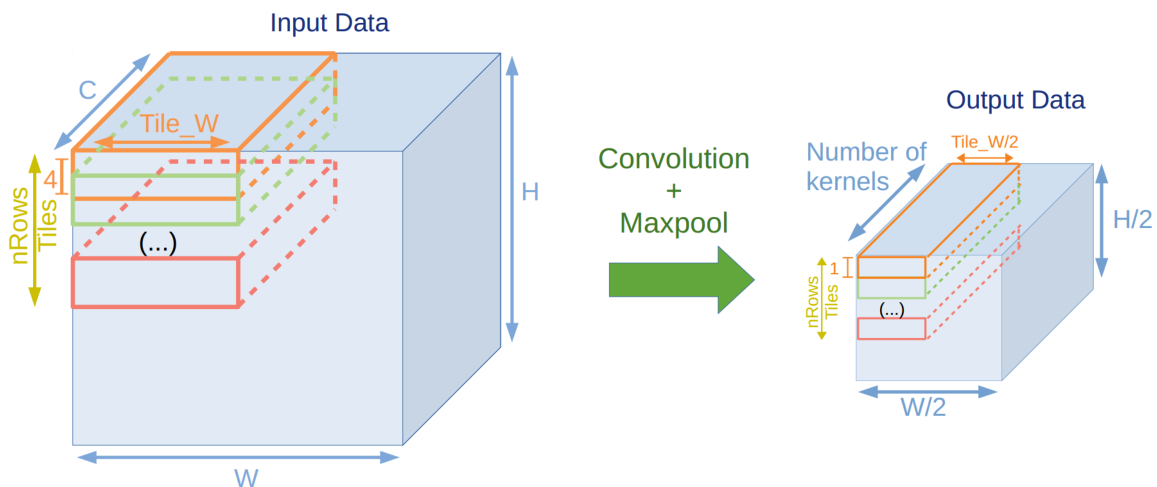


Figure 6.3: Data tiling diagram.

The tiling implemented divides the IFM into blocks of 4 rows, $\texttt{Tile_W} + 2$ columns and $\texttt{C}$ channels. The two extra columns per tile are used as padding to maintain the pattern for convolution on the tile

```
// Fixed configurations
versat.versatRead.setIntPer(ker_side*ker_side*C/nMACs);
versat.versatRead.setIntIncr(1);
versat.versatRead.setIntIter(2*Tile_W);
versat.versatRead.setIntShift(-ker_side*ker_side*C/nMACs);
// Variable configurations
for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles
  for(j=0;j<W/Tile_W;j++){ // for each input tile in a row
    for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
      versat.yread.setIntStart((ker_side*ker_side*C)*l/nMACs);
      versat.yread.setBiasIntStart(l);
    }
  }
}
```

Listing 6.5: Weights and bias configurations for the *Compute* phase

edges. The padded columns are not represented in figure 6.3 for readability purposes. Since there are nRows vReads to store IFM tiles, the same amount of IFM tiles are transferred from main memory.

With the data in ZXY format, each DMA burst reads $\texttt{C} \times (\texttt{Tile_W}+2)$ contiguous values. Each IFM tile starts two rows after the start of the previous tile. The setExtPer() and setExtIncr() configurations follow the same logic used for the kernel configuration. Note that an IFM tile transfer requires $\texttt{ker_side}+1 = 4$ bursts, one for each tile row. After each DMA burst, the shift configuration set in setExtShift() updates the pointer to main memory from the end of the tile row to the start of the next tile row. The vRead for IFM tiles operates in ping-pong fashion: the half of the vRead memory used to write data from main memory toggles between two consecutive datapath phase executions.

The setExtAddr() method updates the pointer to the IFM tile start. The IFM_ptr points to the IFM start. The $k \times 2((\texttt{W}+2) \times \texttt{C} \times \texttt{nRows}$ term accounts for the sets of IFM tile rows already processed, while the $j \times \texttt{Tile_W} \times \texttt{C}$ term accounts for the IFM tiles processed in the current row of tiles. Both terms are multiplied by 2, since each value is 2-byte sized.

6.3.1.2 Compute Phase Configurations

Listing 6.5 presents the configurations to read the weights and biases from vReads into the custom FUs.

The read ports on the weights vReads output nMACs weights in parallel, while the bias vRead outputs nCols values simultaneously.

The vRead for weights is read in sequence, taking $\texttt{ker_side} \times \texttt{ker_side} \times \texttt{C}/\texttt{nMACs}$ operations to read a kernel. The kernels are read $2 \times \texttt{Tile_W}$ times, one per each convolution done over an IFM tile. After each convolution, the internal AGU points back to the initial address, so the setIntShift() configuration is symmetric to the setIntPer() configuration.

The starting address both for reading weights and biases vReads is updated according to each set of kernels being used for convolution.

Listing 6.6 details the configurations to read the IFM values from vReads into the custom FUs.

```
// Fixed configurations
versat.versatComp.read.setIntPer(ker_side*C/nMACs);
versat.versatComp.read.setIntIncr(1);
versat.versatComp.read.setIntIter(ker_side);
versat.versatComp.read.setIntShift((Tile_W+2)*C/nMACs-ker_side*C/nMACs
    );
versat.versatComp.read.setIntIncr2(C/nMACs);
versat.versatComp.read.setIntPer2(2);
versat.versatComp.read.setIntIter2(2);
versat.versatComp.read.setIntShift2((Tile_W+2)*C/nMACs-2*C/nMACs);
versat.versatComp.read.setIntPer3(Tile_W/2);
versat.versatComp.read.setIntIncr3(2*C/nMACs);
versat.versatComp.read.setIntIter3(1);
```

Listing 6.6: IFM tile configurations for the *Compute* phase

```
// Fixed configurations
versat.versatComp.FUComp.setIter(2*Tile_W);
versat.versatComp.FUComp.setPer(ker_side*ker_side*C/nMACs);
versat.versatComp.FUComp.setShift(result_shift);
versat.versatComp.FUComp.setBiasShift(bias_shift);
versat.versatComp.FUComp.setBias(1);
versat.versatComp.FUComp.setLeaky(1);
versat.versatComp.FUComp.setMaxpool(1);
```

Listing 6.7: Custom FU configurations for the *Compute* phase

The IFM tile configurations for the *Compute* phase are constant since the accelerator only operates over one set of IFM tiles at a time. The internal AGU that controls the vRead accesses is configured to generate a five-loop pattern.

The inner-most loop iterates over the values in the same row across all channels for a total of $\texttt{ker_side} \times \texttt{C}/\texttt{nMACs}$. Like the vReads for weights, the vReads for IFM values output nMACs values in the same word. The next loop iterates over the ker_side rows of the input values. The setIntShift() offsets the address to the start of the next row. The execution of these two loops perform a single convolution.

The third and fourth loops generate the pattern for the 2×2 maxpool. The third loop offsets the address to perform the convolution in the next column ($\texttt{C}/\texttt{nMACs}$ addresses added). The fourth loop offsets the address to start in the next row (adding $(\texttt{Tile_W+2}) \times \texttt{C}/\texttt{nmACs}$ to the address) and one column to the left (subtracting $2 \times \texttt{C}/\texttt{nMACs}$).

The fifth loop generates the pattern to move accross the $\texttt{Tile_W}/2$ output 2×2 squares of the IFM tile by adding offseting the address two columns ($2 \times \texttt{C}/\texttt{nMACs}$) to the left.

Listing 6.7 presents the configurations for the custom FU.

The custom FU takes $\texttt{ker_side} \times \texttt{ker_side} \times \texttt{C}/\texttt{nMACs}$ iterations for each convolution, for a total of $2 \times \texttt{Tile_W}$ convolutions per IFM tile. The convolution result and bias values are shifted to comply with the DFP quantization established in table 4.2. The remaining configurations are set according to table 5.3.

Listing 6.8 presents the configurations to write the OFM tiles into the vWrite memory buffers.

```
// Fixed configurations
versat.versatComp.write.setIntDuty(1);
versat.versatComp.write.setIntDelay(READ_LATENCY + WRITE_LATENCY);
versat.versatComp.write.setIntPer(4*ker_side*ker_side*C/nMACs);
versat.versatComp.write.setIntIncr(n_kernels/nCols);
versat.versatComp.write.setIntIter(Tile_W/2);
// Variable configurations
for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles
  for(j=0;j<W/Tile_W;j++){ // for each input tile in a row
    for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
      versat.ywrite.write.setIntStart(l);
    }
  }
}
```

Listing 6.8: vWrite configurations for the *Compute* phase

```
// Fixed configurations
versat.dma.ywrite_write_setNBytesW(Tile_W*n_kernels);
versat.versatComp.write.setOffset(2*((W/2+2)*n_kernels));
versat.versatComp.write.setExtPer(1);
// Variable configurations
for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles
  for(j=0;j<W/Tile_W;j++){ // for each input tile in a row
    for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
      if(l == n_kernels/nCols-1) {
        versat.ywrite.write.setExtAddr(OFM_ptr+2*(k*(W/2+2)*n_kernels*
            nRows+j*(Tile_W/2*n_kernels)));
        versat.ywrite.write.setExtIter(((Tile_W/2)*(n_kernels/nCols)))
            ;
      } else versat.ywrite.write.setExtIter(0);
    }
  }
}
```

Listing 6.9: vWrite configurations for the *Memory Write* phase

The vWrite buffers are written once (`setIntDuty(1)`) for every four convolutions (set by `setIntPer()`). The internal AGU starts the address generation after a delay that accounts for the read process from the vRead memories and the custom FU latency (configured by the `setIntDelay()` method).

Each vWrite receives `nCols` values in parallel, one from each custom FU in the associated row (check figure 5.1). The address offset between each write operation is `n_kernels/nCols`, while the starting address changes for each set of kernels. With this configuration, the results from each set of kernels are intercalated and therefore maintain the ZXY format for the output data. The pattern is repeated for the OFM tile size (`Tile_W/2`).

6.3.1.3 Memory Write Phase Configurations

Listing 6.9 details the configuration to write the OFM tile into main memory.

The DMA is configured with the `setNBytesW()` method that sets the number of bytes per burst. Each burst is sent with an offset of one OFM row ($2(\texttt{W/2+2} \times \texttt{n_kernels}$ bytes).

The OFM tile is only written after all sets of kernels are computed, so the `setExtIter()` is configured at the last set of kernels with the number of vWrite words occupied by the OFM tile. The `OFM_ptr` variable points to the OFM start. The $k \times (\texttt{W}/2+2) \times \texttt{n_kernelsnRows}$ term accounts for the sets of rows of IFM tiles already completed, while the $j \times (\texttt{Tile_W}/2 \times \texttt{n_kernels})$ accounts for the IFM tiles already completed in the current row of IFM tiles.

6.3.2 Other Convolutional Layers

The remaining Convolutional layers of Tiny YOLOv3 are accelerated in isolation (only Convolutional layer) or combined with a Yolo or Upsample layer.

Figure 6.4 presents the tiling format for the remaining Convolutional layers. After layer 8, the IFMs are either 26×26 or 13×13 and therefore $\texttt{Tile_W} = \texttt{W}$. Without tiling across columns, the VersatCNN configurations do not require the second loop present in listing 6.2.

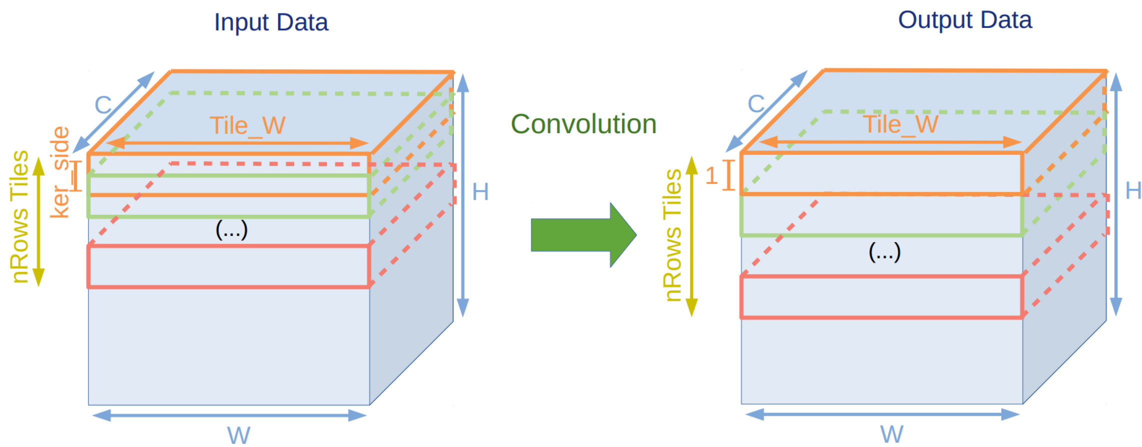


Figure 6.4: Data tiling diagram after layer 8.

The main difference to configurations happens for the vReads of the IFM tiles in the `Memory Read` phase. Listing 6.10 presents the configurations for the Convolutional layer acceleration.

With $\texttt{Tile_W} = \texttt{W}$, the data for the `nRows` IFM tiles read from main memory is in a block of sequential data. Therefore, the data transfer can be configured as a single burst. Note that the AXI4 specification [40] defines a burst size limit of 256, so the operation is divided into maximum-sized bursts in hardware.

After layer 8, the Convolutional layers use either $(1 \times 1 \times \texttt{C})$ or $(3 \times 3 \times \texttt{C})$ kernels. The kernels with $\texttt{ker_side} = 3$ require padding for the rows and columns, which is translated by the `inpadd` flag. For each layer, the set of IFM tiles has $\texttt{C} \times (\texttt{W} + 2 \times \texttt{inpadd}) \times (\texttt{nRows} + 2 \times \texttt{inpadd})$ values. The remaining fixed configurations follow the same pattern from listing 6.4. The `ping_pong` flag enables the ping-pong operation for the IFM tile vReads.

```
1   // Fixed configurations
2   versat.dma.ywrite_read_setLen(C*(W+2*inpadd)*(nRows+2*inpadd)/16-1);
3   versat.ywrite.read.setOffset(2*((W+2*inpadd)*C));
4   versat.ywrite.read.setExtPer((C*(W+2*inpadd)*ker_side)/16);
5   versat.ywrite.read.setExtIncr(16);
6   versat.ywrite.read.setPingPong(ping_pong);
7
8   // Variable configurations
9   for(k = 0; k < W/nRows; k++) { // for each set of rows of input tiles
10    versat.ywrite.read.setExtIter(1);
11    versat.ywrite.read.setExtAddr(IFM_ptr + 2*(k*(W+2*inpadd)*C*nRows));
12    for(l = 0; l < n_kernels/nCols; l++) { //for each set of kernels
13      // versat.run()
14      versat.ywrite.read.setExtIter(0);
15    }
16  }
```

Listing 6.10: Convolutional-only vWrite configurations for the *Memory Write* phase

For each set of the IFM tiles, the external AGU is enabled initialy. Note the `setExtIter()` value of 1 instead of `ker_side`, since the data is read in a single burst. The main memory address is updated with the $2 \times (k \times (\texttt{W} + 2 \times \texttt{inpadd})\texttt{C} \times \texttt{nRows})$ term that accounts for the offset of the IFM tile sets already computed. The external AGU is disabled with `setExtIter(0)` after the first dataflow execution phase of each IFM tile set.

The Convolutional layers 16 and 23 are paired and executed with the Yolo layers 17 and 24 respectively in the same datapath configuration. The Convolutional layers 16 and 23 have linear activation and the Yolo layer operations are equivalent to performing sigmoid activation to selected input channels. Combining both layers consists of performing a regular Convolutional layer with sigmoid activation.

Listing 6.11 presents the custom FU configurations for the execution of the Convolutional and Yolo layer pairs.

The custom FU is configured with `setSigmoid(1)` which enables sigmoid activation, as presented in table 5.4. The custom FU allows for the configuration of a mask (`setSigMask()`) that controls which custom FUs columns execute the sigmoid activation.

The Yolo layers in Tiny YOLOv3 perform sigmoid activation in all channels, except for channels 2, 3, 87, 88, 172 and 173, as shown in table 2.1. A kernel set `l` requires a special sigmoid mask if

$$\texttt{l} == \left\lfloor \frac{\texttt{channel_exception}}{\texttt{nCols}} \right\rfloor \tag{6.2}$$

is true for any of the channel exceptions. Listing 6.12 presents the code for the sigmoid mask calculation.

The `FULL_MASK` is a value with the first `nCols` bits set to 1, one for each kernel computed in the set. The `calc_mask()` iterates over all channels and checks which belong to the current kernel set being used. The position within the kernel set is calculated and the corresponding sigmoid mask bit is set to 0.

The Convolutional layer 19 is paired with the following Upsample layer. Figure 6.5 summarizes the upsampling process from the OFM tile calculated by the regular convolutional configuration.

```
// CustomFU configurations
  // Fixed configurations
  versat.versatComp.FUComp.setIter(W);
  versat.versatComp.FUComp.setPer(ker_side*ker_side*C/nMACs);
  versat.versatComp.FUComp.setShift(result_shift);
  versat.versatComp.FUComp.setBiasShift(bias_shift);
  versat.versatComp.FUComp.setBias(1);
  versat.versatComp.FUComp.setSigmoid(1);
  // Variable configurations
  for(k=0;k< W/(2*nRows);k++){ // for each set of rows of input tiles
    for(l=0;l<n_kernels/nCols;l++){ //for each set of kernels
      // set sigmoid mask
      if(special_kernel_set(l))
        versat.versatComp.FUComp.setSigMask(calc_mask(l));
      else
        versat.versatComp.FUComp.setSigMask(FULL_MASK);
    }
  }
```

Listing 6.11: Custom FU configurations for the Convolutional and Yolo layer pairs

```
  int calc_mask(int l){
    int n_channels = 6;
    int channel_exception[6] = {2, 3, 87, 88, 172, 173};
    int FULL_MASK = (1<<nCols)-1; //all nCols columns enabled
    for(c = 0; c<n_channels;c++){ // for each channel exception
      ch = channel_exception[c];      //get channel value
      ch_pos = (ch-l*nCols); //channel position in current kernel set
      //if channel is in current kernel set
      if( (ch_pos)>0 && (ch_pos)<nCols)
      FULL_MASK -= (1<<(ch_pos)); //remove column from mask
    }
    return FULL_MASK;
  }
```

Listing 6.12: Sigmoid mask calculation

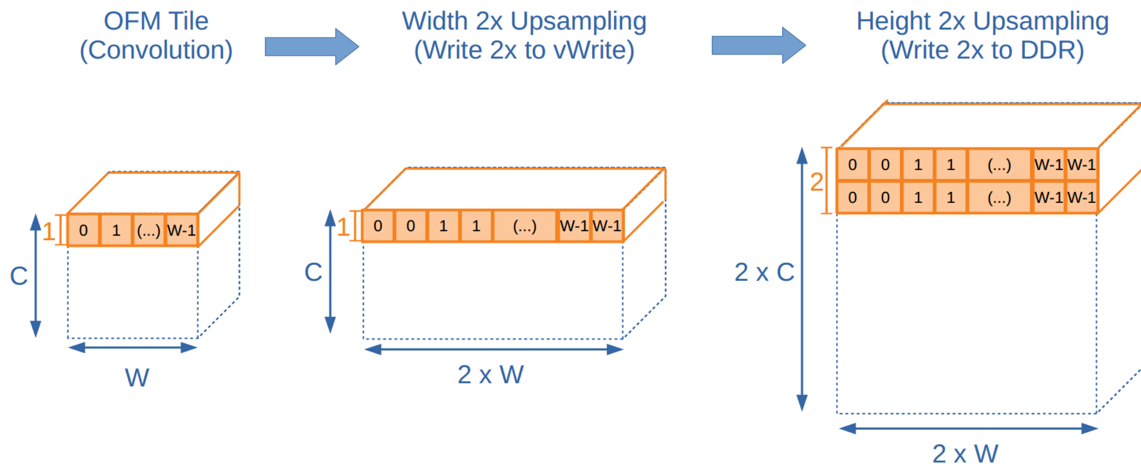


Figure 6.5: Upsampling after convolution diagram.

The vWrite internal AGU is configured with `setIntDuty(2)` to store each convolutional result twice, which automatically performs the width upsampling.

The height upsampling is done by configuring the vWrite external AGU with `setExtIter(2)`, which writes each OFM tile twice to main memory. These changes require updating the remaining configurations with a factor of 2 to account for the upsampled OFM tile row. The `setExtAddr()` configuration needs to account for the factor of 4 from both the width and heigh upsampling.

6.3.3 Maxpool Layer

The Maxpool layers 10 and 12 are accelerated in isolation. Figure 6.6 presents a diagram for the input and output tiles used in each configured dataflow.

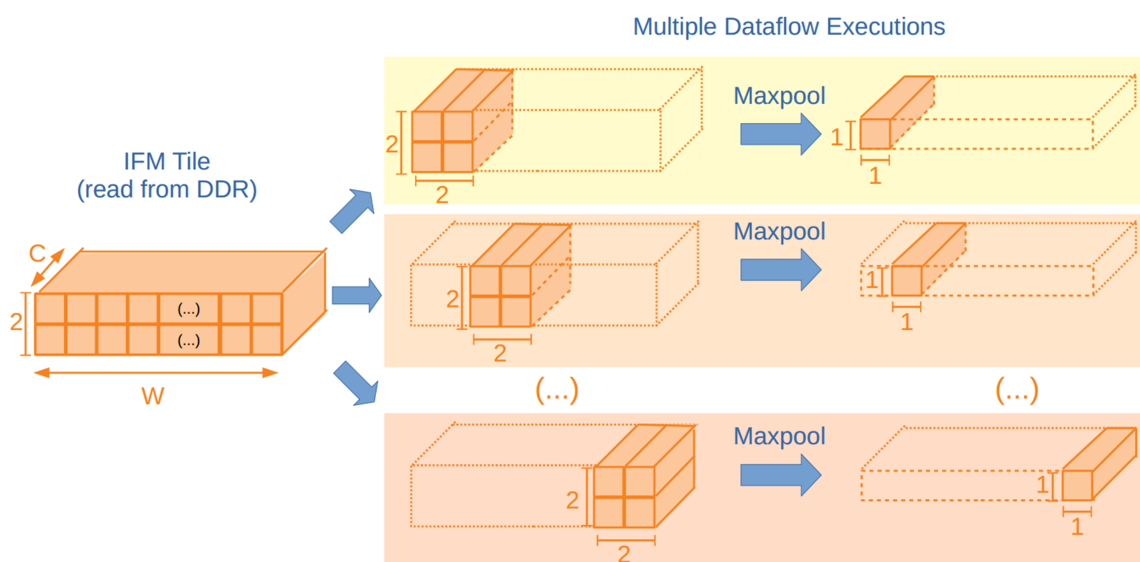


Figure 6.6: Input and output tiles for the Maxpool layer dataflows.

The first dataflow reads the complete layer input into the IFM tile vReads. Each vRead has a $2 \times \mathtt{W} \times \mathtt{C}$ IFM tile. Each dataflow performs maxpooling over a $2 \times 2 \times \mathtt{C}$ input block and outputs a $1 \times 1 \times \mathtt{C}$ output block. Each output block is written to main memory at the end of the dataflow configuration.

Each VersatCNN row performs the maxpooling of a 2×2 region at a time, therefore there is a parallelism of `nRows` for the maxpool acceleration.

The input channels are divided in groups of `nCols` consecutive channels. Each custom FU in a row calculates one of the `nCols` channels. Since a custom FU result is generated at a time for each row, the vWrite write strobe is controlled in hardware to allow writing different 16-bit word into the same vWrite address. Table 5.4 presents the custom FU configuration used.

6.4 Pre-CNN Processing

The CNN pre-processing consists in resizing the input image with $\mathtt{IMG_W} \times \mathtt{IMG_H}$ resolution into an image with the neural network input resolution (416×416).

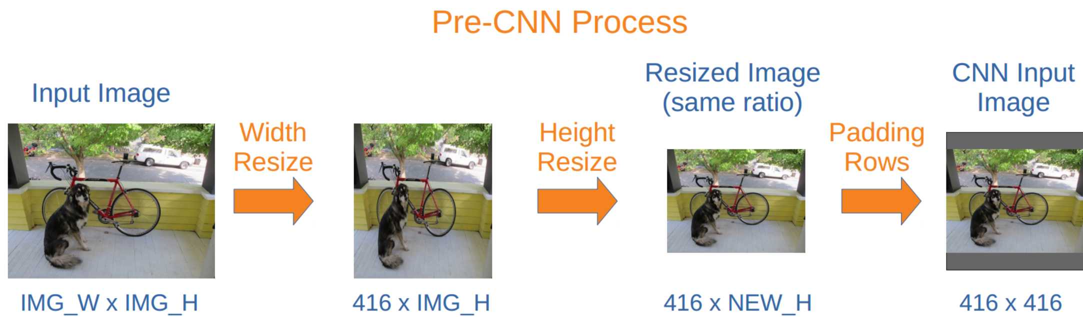


Figure 6.7: Pre-CNN resizing process.

The resizing is done using bilinear interpolation. The bilinear interpolation performs two linear approximations over the image: one for width resizing and another for height resizing. The resized image retains the original aspect ratio, therefore the resized image height is given by

$$\mathtt{NEW_H} = \frac{\mathtt{IMG_H} \times \mathtt{NEW_W}}{\mathtt{IMG_W}}. \tag{6.3}$$

To achieve the target resolution, $(416 - \mathtt{NEW_H})$ padding rows are added to the resized image. Listing 6.13 presents the function for the index pattern and factor calculations.

The process starts by generating a pattern for accessing the resize input images and the respective multiplying factors used in the linear approximation.

The access patterns cannot be mapped into an AGU configuration. Therefore the pattern is calculated by the CPU before the application execution. During the pre-CNN process, the pattern is loaded into a memory used to store patterns for accessing the IFM tile vReads, enabling the acceleration of the width and height resize loops.

```
void preCNN_generate_indexes_factors(int IMG_H, int IMG_W, int NEW_W){
  // resize ratios
  int NEW_H = (IMG_H*NEW_W)/IMG_W;
  float w_scale = (IMG_W-1)/(NEW_W-1);
  float h_scale = (IMG_H-1)/(NEW_H-1);
  // image arrays
  int interm_img[IMG_H][NEW_W] = {0};
  int out_img[NEW_H][NEW_W] = {0};
  // index and factor vectors
  int w_pos[NEW_W] = {0}, w_factor[NEW_W];
  int h_pos[NEW_H] = {0}, h_factor[NEW_H];
  // horizontal indexes and factors
  for(i=0;i<NEW_W;i++){
    w_pos[i] = floor(i*w_scale);
    w_factor[i] = i*w_scale - floor(i*w_scale);
  }
  // vertical indexes and factors
  for(i=0;i<NEW_H,i++){
    h_pos[i] = floor(i*h_scale);
    h_factor[i] = i*h_scale - floor(i*h_scale);
  }
}
```

Listing 6.13: Index pattern and factor calculation

For the width resize loop, each IFM vRead receives an input image row and the weight vReads receive the resize factors. Each custom FU row calculates all image channels in parallel, for a total parallelism factor of $\texttt{nRows} \times 4$, where one of the channels is used for padding.

In the width resize loop, only the first row is used for computation. This limitation is associated with the irregularity of the row indexing across the image, that is not compatible with the regular offset used to load different IFM tiles from main memory. The first row computes all image channels in parallel. Despite the parallelism factor of 4, the width resize still benefits from VersatCNN acceleration since the operation requires large amounts of data transfers that are speedup by the DMA access.

6.5 Post-CNN Processing

The CNN post-processing part of the application, presented in figure 6.1, is composed of three different functions.

Table 6.1 presents the detailed performance of each CNN post-processing function.

Post-CNN Function	Execution Time (ms)
Create boxes	1.94
Filter boxes	0.14
Draw detections	11.91
Total	13.96

Table 6.1: Post-CNN performance using baseline system.

As discussed in section 4.3, the CNN post-processing requires acceleration. The functions to create and filter boxes represent a small portion of the execution time. These functions have a workflow dependent on control instructions which are not suited for acceleration using the VersatCNN CGRA.

The function to draw detections takes 85% of the processing time. Furthermore, this function presents a regular workflow that can be mapped into the VersatCNN.

The drawing of each detection can be accelerated in two routines: one for drawing the class label and another to draw the detection box around the object.

The class labels are received with the weights file and are stored in main memory. The class labels are small images in greyscale that contain the text for each class label. There is an associated colour for all detection labels and boxes drawn of each class. The class label, in grayscale, is multiplied by the class colour to generate the class label with the correct colour. Figure 6.8 presents the class label colouring operation.

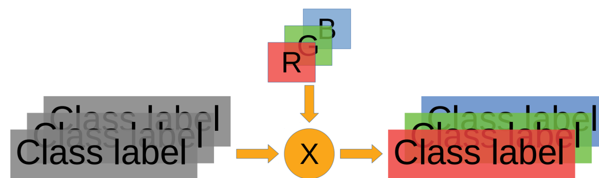


Figure 6.8: Changing label to class colour.

The drawing of the class labels is accelerated in VersatCNN by loading the class labels into the IFM tile vReads and the respective class colour into the weights vReads. The custom FUs execute the MAC output function. The final result is written directly over the input image. The process is repeated for each row of the class label image.

The process for drawing the detection box requires writing the correct class colour into the input images. The process is divided into two distinct sets of configurations.

The first configuration writes the horizontal sides of the detection box. Writing the horizontal sides requires only $2 \times \texttt{Box_side_width}$ VersatCNN executions, since the input image is in ZXY format and all pixels are in adjacent memory positions.

The vertical sides require two write operations each, for a total of $\texttt{Box_height} - 2 \times \texttt{Box_side_width}$ rows. Writing the vertical sides using VersatCNN is still expected to increase the performance versus the baseline system since each write operation writes $\texttt{IMG}_{\texttt{C}} \times \texttt{Box_side_width}$ values to main memory, whereas the cache system only allows writing one value a time.

6.6 Conclusions

This chapter presents the global application from a software perspective. For the CNN pre and post-processing parts, the algorithms are presented and the respective VersatCNN configuration is explained. The CNN acceleration requires different strategies for each layer type. Furthermore, the acceleration of Convolutional layers also requires multiple strategies to account for different input and output sizes

and other special cases in the network. The range of use cases implemented through VersatCNN configurations supports the CGRA as a valid architecture for CNN acceleration. The developed strategies for the Convolutional layers have a parallelism factor of $\texttt{nRows} \times \texttt{nCols} \times \texttt{nMACs}$, while using an efficient pattern for data transfers. The pre and post-CNN processing and the remaining CNN layers have small computational requirements and therefore the strategies developed focus on the efficiency of the data transfers.

Chapter 7

Results

This chapter presents performance results for the IOb-SoC-Yolo system running the Tiny YOLOv3 algorithm. Section 7.1 briefly introduces the IOb-SoC-Yolo implementation used for acceleration. Section 7.2 compares the Tiny YOLOv3 application performance on the following platforms: desktop CPU, GPU and RISC-V soft CPU core running on FPGA. Section 7.3 presents the resources used by the final system. Section 7.4 compares IOb-SoC-Yolo with other FPGA implementations of the Tiny YOLOv3 network.

7.1 IOb-SoC-Yolo System

The IOb-SoC-Yolo system uses the IOb-SoC platform with the VersatCNN CGRA to accelerate the Tiny YOLOv3 application. Table 7.1 lists the parameters for the final VersatCNN version used.

Parameter	Value
nCols	16
nRows	13
nMACs	4
DDR_ADDR_W	32
DATAPATH_W	16
VREAD_BIAS_ADDR_W	3
VREAD_WEIGHT_ADDR_W	14
VREAD_TILE_ADDR_W	15
VREAD_PATTERN_ADDR_W	10
VREAD_TILE_EXT_ADDR_W	15
VWRITE_ADDR_W	8

Table 7.1: VersatCNN parameters for the IOb-SoC-Yolo implementation.

With the chosen paramteres, the VersatCNN is capable of up to $16 \times 13 \times 4 = 832$ parallel MAC operations.

The system is synthesized for a Xilinx XCKU040 FPGA with a clock frequency of 143 MHz.

7.2 Tiny YOLOv3 Acceleration

Table 7.2 presents the Tiny YOLOv3 network execution times on multiple platforms. For each platform, both the CNN function and full application execution times are measured. The full application includes pre and post-processing in addition to the CNN function.

The CPU and GPU results have been obtained using the original Tiny YOLOv3 network [41]. The FPGA results use the embedded software version discussed in Chapter 4.

Software Version	Platform	CNN (ms)	Full Application (ms)	FPS
Floating-Point	CPU (Intel i7-8700 @ 3.2 GHz)	819.2	828.3	1.2
Floating-Point	GPU (RTX 2080ti)	7.5	15.4	65.0
Fixed-Point	FPGA IOb-SoC-Yolo (RISC-V only)	967745.6	968799.5	0.0
Fixed-Point	**FPGA IOb-SoC-Yolo**	**24.4**	**30.9**	**32.4**

Table 7.2: Tiny YOLOv3 execution times on multiple platforms.

The Tiny YOLOv3 on desktop CPUs is too slow for real-time inference. The inference time on an RTX 2080ti GPU shows a 109 speedup versus desktop CPU. On the CPU, the CNN kernel takes about 99% of the total execution time. With GPU acceleration, the CNN kernel execution time drops to under 50% of the total execution time.

Running the complete application on the IOb-SoC-Yolo system, using only the RISC-V CPU for computation takes over 16 minutes. Inference takes over 99.8% of the total execution time, as discussed in section 4.3. Like on the desktop CPU application, the CNN is responsible for most of the processing time.

Using the VersatCNN accelerator, the inference time drops to 24.4 ms, which represents 2.26% of the total execution time. Performing inference with the VersatCNN is 39731 times faster than using the RISC-V CPU alone. Since the computational parallelism factor is 832 (number of parallel MAC units), the remaining speedup obtained is explained with the increased data bandwidth.

The profiling results from table 4.6 show that the acceleration of the pre and post-CNN processing parts of the application is required to achieve real-time performance (30 FPS).

Table 7.3 presents the execution times of the pre and post-CNN parts executing on the RISC-V CPU alone and with the accelerated version using the acceleration strategies explained in sections 6.4 and 6.5.

Hardware used for Execution	Pre-CNN Time (ms)	Post-CNN Time (ms)
RISC-V	1040.2	13.6
VersatCNN	2.9	3.2
Speedup	358.5	4.2

Table 7.3: CNN pre and post-processing times.

With hardware acceleration, both the CNN pre and post-processing times are reduced to milliseconds. The acceleration of both parts only uses one line of FUs with a parallelism factor of $\texttt{nYOLOmacs} = 4$.

Therefore, most of the pre-CNN speedup comes from the improved memory bandwidth of using the DMA instead of the RISC-V cache system. The post-CNN acceleration is less pronounced as only one of the four post-CNN functions is accelerated. Furthermore, small bursts write the detection boxes and labels to memory which reduces the efficiency of the DMA transfers.

With the acceleration of CNN pre and post-processing, IOb-SoC-Yolo achieves a real-time performance of over 30 FPS, as shown in table 7.2.

7.3 FPGA Resource Utilization

Table 7.4 presents FPGA resource utilization of the final system for a Xilinx XCKU040 FPGA device of the Kintex UltraScale product family. The IOb-SoC-Yolo column indicates the resources used for the implementation of the IOb-SoC platform, peripherals and VersatCNN CGRA. The column with the total resource utilization also includes other modules outside of the system like the AXI4 interconnect [42] and the DDR4 memory controler [43] from Xilinx.

The design utilizes about 50% of the available Look-Up Tables (LUT) logic, 18% of the look-up tables implementable as distributed RAM (LUTRAM), 46% of the Flip-flops (FF) and 46% of the Digital Signal Processors (DSP). The design uses about 65% of the available 36 kB Block RAMs (BRAM).

The majority of the resources is used to implement the VersatCNN CGRA. The VersatCNN uses 832 DSPs for the custom FU MAC blocks. The remaining DSPs are used for the calculation of the external address offsets for ($\texttt{nCols}-1=15$) weight vReads, ($\texttt{nRows}-1$) IFM tile vReads and ($\texttt{nRows}-1$) vWrites.

Resource	VersatCNN	IOb-SoC-Yolo	Total (Used%)
LUT logic	103655	106892	119166 (49)
LUTRAM	16792	17120	19780 (18)
FFs	86319	88460	110988 (46)
36kB BRAMs	339	356	385 (64)
DSPs	871	875	878 (46)

Table 7.4: IOb-SoC-Yolo resource utilization in a Xilinx XCKU040 FPGA.

7.4 Comparison With Other FPGA Implementations

The only competing Tiny YOLOv3 FPGA implementations known by the author, at the time of writing this document are [44], [45] and [46]. Table 7.5 compares IOb-SoC-Yolo with these works in terms of performance.

The implementation in [44] uses a Zynq 7020 SoC, whose FPGA part has significantly fewer resources than [45] and this work. The Zynq device implementation uses the on-chip dual-core ARM CPU that executes at 667 MHz, while [45] and this work use soft CPU cores with the same frequency as the developed hardware accelerators. [46] uses a Zynq UltraScale+ SoC with an on-chip quad-core ARM CPU capable of up to 1.5GHz clock.

This work, along with [44] and [45] use 16 to 18-bit quantizations, while [46] uses a 8-bit quantization.

Table 7.5: Performance comparison with other FPGA implementations.

	[46]	[44]	[45]	IOb-SoC-Yolo
Device	Zynq UltraScale+ XCZU9EG	Zynq 7020	Virtex-7 XC7VX485T	UltraScale XCKU040
HW Freq. (MHz)	-	100	200	143
CPU Freq. (MHz)	1.5GHz[a]	667	200	143
DSPs (CNN only)	-	120	2304	878 (832)
GMAC/s	290.3	5.2	230.4[b]	114.2
MMAC/s/DSP	-	33	100	137
CNN Exec. (ms)	9.6	532.0	-	24.4
Quantization	8-bit	16-bit	18-bit	16-bit

[a] Maximum clock for the device [47].
[b] Peak performance.

The number of DSP blocks used in [44] is about $7\times$ lower than those used in IOb-SoC-Yolo, while [45] uses over $2.5\times$ more DSP blocks than IOb-SoC-Yolo, which shows the number of DSP blocks both for the complete system and the inference accelerator.

The MMAC/s figure measures the number of multiply+accumulate operations performed per second by each of the works. For [44] and this work, the MMAC/s figure is obtained by

$$\texttt{MMAC/s} = \frac{\texttt{\#CNN MACs}}{\texttt{CNN Execution Time (s)}}. \tag{7.1}$$

The Tiny YOLOv3 network has a total workload of 2.78 GMAC operations. Only the Convolutional layer MACs are accounted to calculate the workload. The GMAC/s figure reported in [45] is obtained by

$$\texttt{GMAC/s} = \texttt{\#DSP} \times \texttt{Freq}. \tag{7.2}$$

This value represents the maximum achievable operations per second rate of the design. The real throughput of the [45] implementation is expected to be lower.

The DSP efficiency figure is presented to account for the different scale of resources used in the different implementations. This work presents over $4\times$ more DSP efficiency than [44]. Even when compared with the maximum achievable throughput in [45], this work obtains a higher DSP efficiency. This is expected as [44] and [45] only perform acceleration for the Convolutional layers, while this work accelerates all network layers.

[46] claims a performance of over 100 FPS using a software and hardware co-design approach. The Convolutional, Maxpool and Upsample layers are accelerated in programmable logic. The resource usage and the clock frequency of the final system is unknown. The pre and post-processing parts of the application are executed in the processing system of the device.

Notice that this performance analysis encompasses only the neural network execution. The CNN pre and post-processing parts of the Tiny YOLOv3 application are outside the scope of all the other works presented. Although the CNN part is responsible for most of the computational load, the pre and post-processing parts become the bottleneck for real-time execution, as shown in section 4.3.

7.5 Conclusions

In this chapter, performance results for the Tiny YOLOv3 application running on the IOb-SoC-Yolo platform have been presented and compared with desktop CPU and GPU execution results. With massive acceleration of the CNN kernel, and by also accelerating the pre and post-processing parts of the application, it was possible to achieve real-time execution of over 30 frames per second. The designed strategies focussed on efficient data transfers enabled most of the speedup outside of convolunional acceleration. The resource utilization results for the target FPGA device provides evidence that the proposed implementation is relatively economical on hardware resources while showing possibilities for higher parallelism using the same device. The developed system obtains better DSP efficiency when compared to other competing Tiny YOLOv3 FPGA implementations.

Chapter 8

Conclusions

This work describes the development of a complete embedded system application to execute on the VersatCNN CGRA accelerator that enables real-time (30 FPS) performance for the Tiny YOLOv3 neural network. In particular, focuses on the design of configurations strategies for the VersatCNN CGRA to enable acceleration of the CNN layers and other routines.

The software model uses a 16-bit fixed-point representation, suited for embedded execution, for the neural network values, quantized from the original floating-point values. The batch normalize weights are fused with the convolutional weights to reduce the number of weights and computation. The leaky and sigmoid activations are approximate, using sums of powers of two. Dynamic fixed-point quantization is applied to improve the model's mAP. The final neural network model scores 30.8 $\mathtt{mAP}_{50}$ in the COCO 2017 test dataset, while the original model scores 32.9 $\mathtt{mAP}_{50}$. Given the drastic simplification achieved with the fixed-point representation, this accuracy loss is very acceptable.

The IOb-SoC platform uses a RISC-V soft CPU core that has access to static on-chip memory and DDR external memory. The system also has a UART peripheral used to load the application firmware and print debug messages on the user's host PC. To establish an effective development and testing environment, this work integrated additional timer and ethernet modules as system peripherals. The timer module measures the system performance at the system clock period resolution. The Ethernet module transfers the CNN weights file and the input and output images. A ethernet communication protocol for data transfers between the embedded device and a computer was developed.

The VersatCNN CGRA is designed for a *memory read* $\rightarrow$ *compute* $\rightarrow$ *memory write* workflow. The use of a DMA module increases the data bandwidth by bypassing the CPU + cache system. A matrix of custom computational FUs massively accelerates the computation. The accelerator explores three types of convolutional parallelism: inter-FM, intra-FM and inter-convolution. The parameterizable accelerator takes advantage of the parallelization factor in all Convolutional layers.

A set of configuration strategies for the embedded software is developed to accelerate the application. The configurations include strategies to accelerate Convolutional layers or Convolutional layers paired with Maxpool, Yolo or Upsample layers. Beyond the CNN inference, the image resizing and the drawing of the final detection boxes and labels are also accelerated to achieve the target performance.

8.1 Achievements

The final system achieves a performance of over 30 FPS to execute the complete Tiny YOLOv3 application. To the best of the author's knowledge, this is the only work that attempts to accelerate the CNN execution by improving all layer types. This work is also the only known work that provides acceleration solutions for the complete Tiny YOLOv3 application beyond CNN inference.

The acceleration of the non-convolutional parts of the application highlights the relevance of the data bandwidth in the final performance. In those cases, the achieved speedup is, at times, orders of magnitude higher than the parallelism factor exploited for computation.

8.2 Future Work

IOb-SoC-Yolo can improve in several ways, according to the specified goal. These goals can include accelerating different CNNs, improving the overall system performance or the method for image input and output.

The VersatCNN CGRA can be validated by accelerating other neural networks. The development of the VersatCNN configurations can be facilitated by a library of generic functions that configure different layer types layers. A next step includes the development of a automatic tool that reads the neural network configuration and the VersatCNN parameters to generate the configuration software.

To improve the overall system performance, one could replace the soft CPU core used in the system. Currently, the system uses the PicoRV32 RISC-V core that needs approximately four clock cycles per instruction (CPI=4). The IOb-SoC platform uses a CPU wrapper to interface any CPU with the rest of the system. A higher performance CPU will accelerate the sequential parts of the application. In the Tiny YOLOv3 application, the CPU is still responsible for configuring the VersatCNN in between runs, and running part of the CNN post-processing functions.

The Ethernet peripheral can transfer large quantities of data in and out of the system at up to 100 Mbps. However, with the current system, the transfers are bottlenecked by data reads and writes to main memory, performed by the cache system. Writes are way more problematic than reads. DMA access to the Ethernet module could improve these data transfers. This improvement would also enable real-time image transfers to and from the system.

Another potential I/O improvement to the system would be the implementation of a HDMI module to connect the system directly to a camera. This improvement would be suitable for deployment on edge processing nodes.

Bibliography

[1] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[2] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015.

[3] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[4] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature Pyramid Networks for Object Detection," *CoRR*, vol. abs/1612.03144, 2016.

[5] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *CoRR*, vol. abs/1506.02640, 2015.

[6] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[7] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[8] J. Redmon, "YOLO: Real-Time Object Detection." `http://pjreddie.com/darknet/yolo/`, visited in 20 Nov 2020, 2018.

[9] D. Pestana, "Object Detection and Classification on the Versat Reconfigurable Processor." Master's Thesis, Jan 2021.

[10] I. Lda, "IOb-SoC." `https://github.com/IObundle/iob-soc`, 2020.

[11] A. Charana, "Development Environment for a RISC-V Processor." Master's Thesis, July 2020.

[12] J. Roque, "Development Environment for a RISC-V Processor: Cache." Master's Thesis, Jan 2021.

[13] V. Mário, "Deep Versat: A Deep Coarse Grain Reconfigurable Array," Master's thesis, Instituto Superior Técnico, November 2019. Master's Thesis.

[14] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, pp. 541–551, Dec 1989.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[16] 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)*, vol. 115, no. 3, pp. 211–252, 2015.

[17] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[18] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *Proceedings of IEEE*, vol. 105, pp. 2295 – 2329, December 2017.

[19] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *CoRR*, vol. abs/1512.03385, 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*, vol. abs/1405.0312, 2014.

[21] A. Vidhya, “Yolov3 theory explained.” `https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193`, Visited in 22 Nov 2019, July 2019.

[22] J. Hui, “mAP (mean Average Precision) for Object Detection.” `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`, Visited in 7 Jan 2020, Mar. 2018.

[23] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection,” *CoRR*, vol. abs/1708.02002, 2017.

[24] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, “Accelerating CNN inference on FPGAs: A Survey,” *CoRR*, vol. abs/1806.01683, 2018.

[25] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, “Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, pp. 1354–1367, July 2018.

[26] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” pp. 26–35, 02 2016.

[27] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks,” pp. 16–25, 02 2016.

[28] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster:,” pp. 326–331, 08 2016.

[29] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv 1409.1556*, 09 2014.

[30] Yufei Ma, N. Suda, Yu Cao, J. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Aug 2016.

[31] M. Lin, Q. Chen, and S. Yan, "Network In Network," 2013.

[32] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning," pp. 6071–6079, 07 2017.

[33] T. Fujii, S. Sato, H. Nakahara, and M. Motomura, "An FPGA Realization of a Deep Convolutional Neural Network Using a Threshold Neuron Pruning," pp. 268–280, 03 2017.

[34] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Hock, Y. Liew, K. Srivatsan, and D. Moss, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?," pp. 5–14, 02 2017.

[35] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 235–244, July 2016.

[36] J. D. Lopes, "Versat, a compile-friendly reconfigurable processor-architecture," Master's thesis, Instituto Superior Técnico, November 2017.

[37] I. Tsmots, O. Skorokhoda, and V. Rabyk, "Hardware Implementation of Sigmoid Activation Functions using FPGA," in *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, pp. 34–38, 2019.

[38] C. Wolf and et. al., "PicoRV32 - A Size-Optimized RISC-V CPU." `https://github.com/cliffordwolf/picorv32`, 2019.

[39] K. Cheng and et. al., "RISC-V GNU Compiler Toolchain." `https://github.com/riscv/riscv-gnu-toolchain`, 2020.

[40] ARM, *AMBA AXI and ACE Protocol Specification*. ARM, February 2013.

[41] J. Redmond, "darknet." `https://github.com/pjreddie/darknet`, 2018.

[42] Xilinx, *AXI Interconnect v2.1*. Xilinx, December 2017.

[43] Xilinx, *UltraScale Architecture-Based FPGAs Memory IP v1.4*. Xilinx, June 2020.

[44] Z. Yu and C. Bouganis, *A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny*, pp. 330–344. 03 2020.

[45] A. Ahmad, M. A. Pasha, and G. J. Raza, "Accelerating Tiny YOLOv3 using FPGA-Based Hardware/-Software Co-Design," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2020.

[46] S. Oh, J. H. You, and Y. K. Kim, "Implementation of Compressed YOLOv3-tiny on FPGA-SoC," in *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, pp. 1–4, 2020.

[47] Xilinx, *Zynq UltraScale+ MPSoC Data Sheet v1.8*. Xilinx, October 2019.