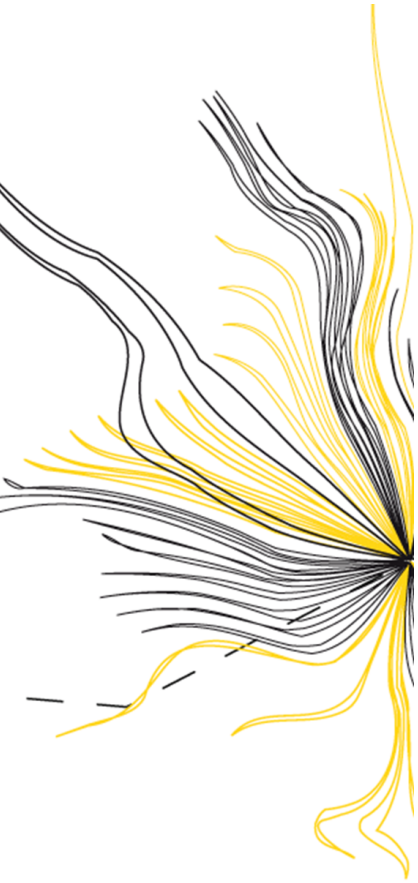
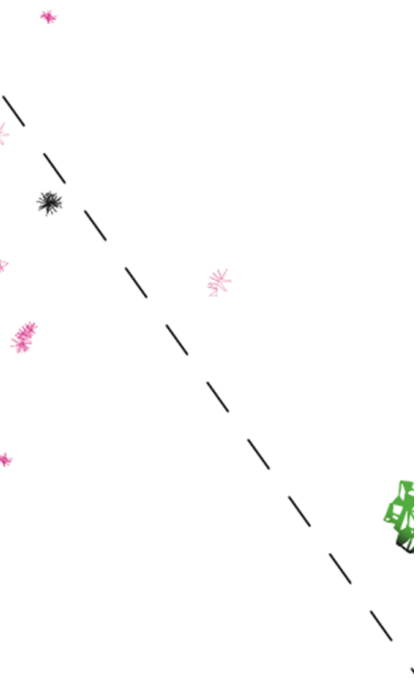


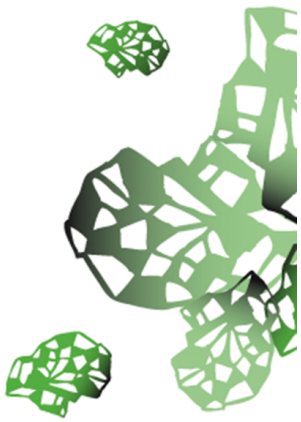
UNIVERSITY OF TWENTE.

CNN Accelerator Throughput Improvement using High-Level Synthesis for FPGA

Matthijs van Minnen

MSc. Thesis
April 2022





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

Computer Architectures and
Embedded Systems Group

Faculty of Electrical Engineering,
Mathematics & Computer Science

University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

ABSTRACT

The number of applications for neural network is growing, which increases the demand for processing power to run these networks. General purpose solutions are available, but specialised hardware can provide better performance at a lower energy cost. An accelerator is developed for FPGA to increase the throughput for the convolutional layers of the YOLOv4 Tiny CNN. Catapult HLS is used to speed up development of the accelerator. Using HLS, a design is developed that is inspired by the Eyeriss architecture. As the tool does not natively infer DSP-blocks in the design, a custom design flow is derived to instantiate these blocks to perform the MACC operations. With this implementation, a MACC operation is performed in 1 clock cycle. A schedule is found to optimise the hardware usage for the given CNN, using the Timeloop tool. This yields 99% utilisation of the hardware. The hardware implementation is simplified to meet the throughput requirements for providing data for the MACC operations. With the optimised schedule and improved hardware, it is estimated that the accelerator provides a throughput of 4GOPS, whilst simultaneously reducing the resource utilisation by ~30%, compared to other works.

LIST OF ACRONYMS

AC	Algorithmic C
AGEN	Address generator
AI	Artificial intelligence
AP	Average precision
ASIC	Application specific integrated circuit
CNN	Convolutional neural network
COCO	Common objects in context
CSC	Compressed sparse column
DPU	Deep learning processing unit
FC	Fully connected
FF	Flip-flop
FLOPS	Floating-point operations per second
FPS	Frames per second
GLB	Global buffer
GMACC	Giga multiply-accumulates
GOPS	Giga operations per second
GUI	Graphical user interface
HDL	Hardware description language
HLS	High-level synthesis
ifmap	Input feature map
IoU	Intersection over union
LN	Local network
LUT	Look-up table
MACC	Multiply-accumulate
mAP	Mean average precision
MC	Multicast controller
NN	Neural network
NoC	Network-on-chip
ofmap	Output feature map
PE	Processing element
PL	Programmable logic
PS	Processing system
psum	Partial sum
ReLU	Rectified linear unit
RLC	Run length compression
RS	Row stationary
RTL	Register transfer level
SoC	System-on-chip
spad	Scratchpad
YOLO	You Only Look Once

CONTENTS

1. INTRODUCTION

Artificial intelligence (AI) is applied to a growing number of applications, trying to make sense of data. The performance of these AI systems can be outstanding after sufficient training, but can also be a high computational burden due to the sheer number of operations required for one inference of the algorithm. Specialised hardware has been used successfully to improve both performance and power requirements over general purpose processors, as specialised hardware can be tailored for a specific set of instructions or operations, but more importantly, can perform computations in parallel.

In the field of image classification, where images are scanned for objects and subsequently scored for a fixed range of classes, networks are generally characterised as *CNNs*. This name is derived from the large number of convolution operations that are performed on the input image with different filter kernels. There are a variety of popular models that are used to perform this image classification: AlexNet, MobileNet [13], ResNet, VGG and YOLO [25].

These networks are ranked based on performance on the one hand, and complexity, the number of computations, on the other. Often times, a higher complexity results in a higher performance. A trend can be observed the last few years, where the performance per complexity metric has improved over time, which becomes even more prevalent for mobile applications with for instance MobileNet.

One kind of CNN is the You Only Look Once (YOLO) network, which has seen multiple versions, the current one being Version 4. Based on this YOLO Version 4, an optimised model has been created focused on memory-limited devices: *YOLOv4 Tiny* [25]. It reduces the memory requirement of the model by scaling down the number of weights in the network. Effectively, the number of computations is reduced from 59.56FLOPS to 6.91FLOPS (floating-point operations per second). This affects the overall performance, when compared to the original YOLOv4. This CNN was the target of an accelerator created by Heinsius [12] to improve the throughput. The accelerator is based on the Eyeriss architecture [7] and was implemented using Catapult [21], a HLS tool, on the Zedboard FPGA platform with a Xilinx Zynq-7020 system-on-chip (SoC). The premise of HLS is that it can accelerate the development of hardware designs, by working at an abstraction level higher than hardware description language (HDL) and enabling higher level simulations. That work accelerates the computations 3.84 times over just the processing system (PS) in the SoC whilst simultaneously reducing power requirements. However, the goal to achieve real-time processing was not achieved.

1.1 Research questions

This research proceeds on the work of Heinsius with the aim of exploring means to improve the throughput. The same boundary conditions will be applied. The CNN to be accelerated is YOLOv4 Tiny and will be implemented using Catapult HLS on the Zedboard. The goal by Heinsius was to deliver real-time performance, defined as 30frames per second (FPS) when processing images. As the absolute performance is largely dependant on the scale of the hardware, the

goal of this work is adjusted to increase the throughput and optimise it for the Zedboard—whilst maintaining a flexible framework that can support a variety of CNN algorithms and frameworks. Optimally, the performance of the new Accelerator should approach the theoretical maximum of the hardware platform. Given this objective, the following research question can be formulated:

How can the throughput of a hardware accelerator for a CNN be improved to approach the theoretical upper bound using HLS for FPGA?

In order to compare the performance against the theoretical limit, this limit must be known. When designed for a specific device, the performance is limited by the available hardware. To stay more general, an analysis is performed which is hardware agnostic. To provide a reference, the design is then synthesised for the Zedboard, as a reference. To derive an upper bound for the performance the following question must be answered: *What is the theoretical upper bound for a CNN accelerator's performance on the Zedboard and how does this compare to performance in literature?*

Heinsius finds several points of improvements to increase the throughput of his design. There is communication between the off-chip memory and the Accelerator as well as an *network-on-chip (NoC)* to distribute data on-chip. Heinsius identifies a bottleneck in these communication networks providing data for the computations. More specifically, the NoC seems to lack the throughput to sustain the rate of computations. To address this, the following sub-research question is formulated: *Can the network-on-chip bandwidth bottleneck in the work by Heinsius be identified and resolved?*

The goal is to utilise HLS to accelerate the development of the new architecture. However, HLS is written at a different abstraction level when compared to HDL. Development times could be shortened and simulations can be performed faster, but performance *may* be impacted. To evaluate the effect of using HLS, the following sub-research question is formulated: *Does a workflow with High-Level Synthesis influence the performance of a CNN accelerator?* In the Accelerator by Heinsius, no DSP-blocks from the FPGA fabric were used to accelerate the MACC operations. Given the computational qualities of the DSP-blocks, it is expected that using their processing power will provide a significantly higher throughput for MACC operations than using look-up tables (LUTs) for these computations. Since they were not implemented in the original Accelerator, a work-around should be found to implement them using HLS. This topic is covered by the following sub-research question: *How can DSP-blocks be implemented in the Accelerator design using the high-level synthesis workflow?*

The research questions are summarised below:

- *How can the throughput of a hardware accelerator for a CNN be improved to approach the theoretical upper bound using HLS for FPGA?*
- What is the theoretical upper bound for a CNN accelerator's performance on the Zedboard and how does this compare to performance in literature?
- Can the network-on-chip bandwidth bottleneck in the work by Heinsius be identified and resolved?
- Does a workflow with high-level synthesis influence the performance of a CNN accelerator?
- How can DSP-blocks be implemented in the Accelerator design using the high-level synthesis workflow?

1.2 Contributions

This work provides a number of aspects to improve the throughput of the CNN accelerator. To speed up the MACC computations, DSP-blocks are implemented in the design. This occurs outside of Catapult, as the tool is not able to automatically infer these blocks for all FPGA families. To achieve this, the processing element (PE) will be implemented using a black-box structure in Catapult, such that it can later be substituted for an IP-block (the DSP48E1 primitive in case of the Zynq 7020 fabric).

Moreover, an analysis is performed of the throughput requirements of the NoC, which provides data to the PE-array. Using the Timeloop tool, an optimal schedule is derived which is optimal for the hardware. With this, the design of the array is simplified to be 1 dimensional and optimised based on the found requirements. This uses the hardware more efficiently, yielding better performance at a lower hardware cost. As hardware is freed up, it can be re-deployed to perform more computations in parallel, yielding even higher throughput.

1.3 Outline

To get acquainted with the topic of CNNs and the relevant aspects for creating hardware accelerators, Chapter 2 discusses the theoretical background and will introduce relevant terminology. It will also discuss the properties of YOLOv4 Tiny and how these are relevant for the Accelerator. Finally, the operation of the Timeloop tool will be explored such that it can be utilised later. Chapter 3 discusses existing topologies for hardware accelerators and explores their merits. Secondly, similar works also targeting the Zedboard, or more generally the Zynq 7020 FPGA, are summarised to provide a comparison for performance and derive a reasonable upper bound for the performance. Here the work by Heinsius will also be explored in more depth to understand the shortcomings. Chapter 4 explores the Catapult tooling and describes the workflow for setting up an Accelerator using HLS.

In Chapter 5 three contributions will be provided. First, the workflow to implement DSP-blocks in the design is described. Secondly, the PE-array is re-evaluated. To lastly, optimise the throughput of the array. The results of these efforts will be discussed in Chapter 6, along with the presentation of a schedule, derived with the Timeloop tool, that is optimised for the presented architecture and the YOLOv4 Tiny network.

Conclusions will be drawn from these results in Chapter 7 and further discussed in Chapter 8. In this final chapter, possible recommendations and points for improvement will be presented.

2. CONVOLUTIONAL NEURAL NETWORKS

There are many sub-classes of AI networks. To perform object detection in images, CNNs are the prevalent technology. As described, YOLOv4 Tiny will be used to infer the objects. However, to get a better understanding of CNNs, a general description of NNs is provided to discuss the topics relevant for this work. The description will not deal with the topic of object detection or how an algorithm would perform that task. Instead, a more abstract view of the computations is provided, which aids in the development of an Accelerator for those computations.

2.1 Neural network background

Objects are detected in an image by the CNN. It does so by convolving the image with a number of filters. Using these computations, the network distills features from the image as the different filters reveal structures (e.g. edges, textures or colours) in the image and identifies objects based on these structures and classifies the object to one of the available classes, available from a fixed list.

The computations in a NN are derived from the electrochemical processes inside neurons and especially how they are interconnected in the (human) brain. NNs are inspired by the operation of the brain to execute complex tasks, using a mathematical model. The computations in the NN are largely similar to a biological brain, as signals on interconnections from other neurons are gathered in the cell body and propagated. A formal definition of the NN's operation is given as:

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

Here N is the number of interconnections with other neurons, x_i the different inputs and w_i are the weights. In the brain the summed signal should pass a threshold to initiate a chemical reaction, such that the neuron fires. This threshold introduces non-linearity and is also important for the operation of a NN. Without this non-linearity, the network could be rewritten as one large

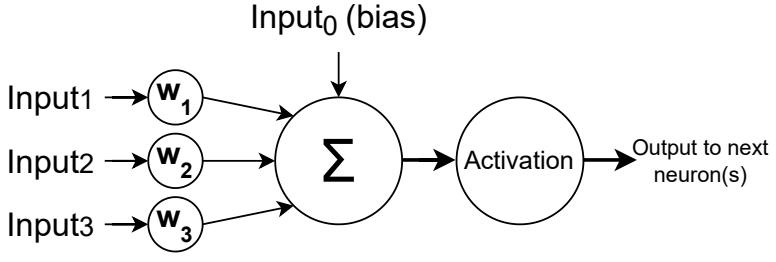


Figure 2.1: Graphical representation of a neuron.

matrix multiplication and ultimately be resolved by multiplying a matrix of inputs with a matrix of filter values. For NNs the firing of the neuron is decided by an activation function, which is applied on the sum of the weighted inputs. In Equation (2.1) $f()$ is this activation function. Figure 2.1 represents the general structure of an artificial neuron, including the multiplication with weighting factors, accumulation and the application of an activation function.

The activation functions are further discussed in Section 2.1.1. The different neurons are connected in layers. Besides the regular neurons described previously, there are also alternative layers which will be discussed in Section 2.1.2. To distinguish different networks they need to be comparable on some metrics, these are discussed in Section 2.1.3.

2.1.1 Activation functions

The concept of a threshold that needs to be exceeded before a neuron fires is modelled using the *rectified linear unit (ReLU)*, which is formally described as:

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{2.2}$$

This activation function is non-linear. From a computational point of view this is beneficial; any future multiplications or additions with this output of the neuron can be omitted, as it has a value of zero. This technique is leveraged in [6]. For a NN, having many neurons with zero as output might starve the network, and stop it from producing a sensible output. Hence, there are alternative activation functions which can be used that do not starve the network. An example is *leaky ReLU* which is similar to ReLU, but does not produce zeros at the output, as it multiplies negative values with a coefficient:

$$f(x) = \begin{cases} \alpha x & \text{for } x \leq 0, \text{ with } 0 < \alpha < 1 \\ x & \text{for } x > 0 \end{cases} \tag{2.3}$$

As a neuron will always produce an output, albeit very small, it means that the provided inputs will always propagate through the network, thus avoiding starvation of the network. There are alternative functions with more complex descriptions, such as the sigmoid $\frac{1}{1+e^{-x}}$ or hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. These last two functions are computationally more intensive but can provide a smoother transition region which might be beneficial for the performance of the network. The different activation functions discussed here are plotted in Figure 2.2. Alternatively there is the Softmax function which is often used in classification layers. It normalises the inputs based on the dimension of the input vector **x** to a probability density function. It is defined as:

$$y(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{dim(\mathbf{x})} e^{x_j}} \tag{2.4}$$

2.1.2 Layer types

A CNN consists of different layers, each with different characteristics. In Section 2.1 the general background was discussed with more information on neurons. These neurons provide the basic building blocks for the most important layers of a CNN. A neuron can have a variable number of inputs which are connected to the outputs of neurons from a previous layer. Three layers types are discussed here: fully connected (FC), convolutional and pooling layers.

A FC layer is also built from neurons and connects *all* neurons from the prior layer to all neurons of the next layer, thus fully connecting both layers. In convolutional layers, a convolution kernel

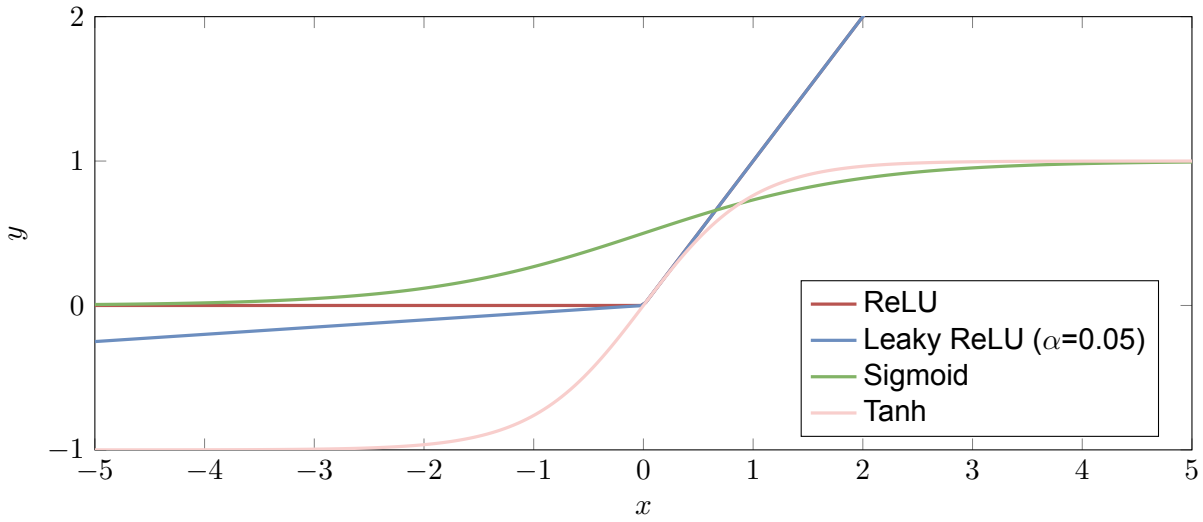


Figure 2.2: Different activation functions for neural networks.

is applied to the inputs (for example, see Figure 2.6). In this case, a neuron computes one output value of the convolution and is only connected to a number of inputs equal to the size of the convolution kernel. A FC layer has a neuron for every neuron in the next layer (N_{out}) and a weight for every input (N_{in}), which results in $N_{in} \times N_{out}$ different weights and multiplications. The number of weights for a convolutional layer is significantly lower as a kernel of dimensions $n \times m$ (usually square $n = m$ and $n \gg N_{in}$ and $m \gg N_{out}$) is applied to all inputs. Applying this kernel to all inputs is realised by reusing the same weights in different neurons. In the process of training a NN, the weights in these layers are adjusted to improve the performance of the entire network. This performance is ranked according to a loss-function. The selection of this function is an important aspect of training a network. With the lower number of weights used in convolutional layers, it becomes easier to train the network, as fewer parameters have to be optimised. FC layers are typically used in the last few layers of a CNN, as the dimensions (N_{in} and N_{out}) are significantly smaller compared to the input of the network.

For object classification networks like YOLO, the FC layer is used to perform the classification. The output of the classification network provides a decision if an object belongs to a number of fixed classes, represented by individual neurons. After an input image is processed and reduced using several convolutional layers, it is passed to the FC layer for classification. Next it is weighted by every neuron and an activation function is applied to make a binary decision if an object belongs to a specific class, or a probability for a match is derived using the softmax function.

Another common layer in CNNs is the pooling layer. In contrast to the previously discussed layers, this layer cannot be trained, as it is not built up from neurons and thus does not contain trainable weights. Instead, the pooling layer is used to reduce the dimensions of the layers, effectively downsampling the inputs. By pooling data, multiple adjacent values are pooled to form 1 output. Multiple pooling strategies are possible. Max and average pooling are most common. Max and average pooling respectively find the maximum and average value of all values from an $n \times m$ rectangle and use this to determine the single output value. This is visualised in Figure 2.3, where the rectange is size 2×2. The $n \times m$ rectangle can be used to reduce the dimensions of a 2D plane, but can also be applied to reduce the number of channels of the data, flattening the layer.

An alternative for reducing the dimensions of the data is applying a stride to the inputs. Under normal operation, when the stride is 1, the kernel is moved 1 position at a time over the inputs.

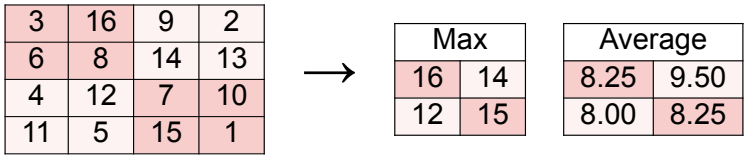


Figure 2.3: An example of max and average pooling applied to a random 4x4 matrix.

When a non-unit stride is applied, the kernel is shifted over the inputs by the stride factor (S), thus moving over S input values. This reduces the amount of times the kernel is multiplied with the input, creating fewer output values. This scales down the output dimensions similar to how a pooling layer would. Typically, a pooling layer is applied directly after a convolutional layer and pooling is thus performed *after* the activation function. When stride is applied, the reduction is performed *before* the activation function. This difference affects the performance of the network. From a hardware point of view, stride is more efficient, as it does not require additional computations to reduce the output dimensions. The pooling layer applies an additional operation (either finds an average or maximum) on the input data, which requires additional hardware resources.

Preferably, stride would be applied over pooling, as it cuts down the number of computations whilst reducing the dimensions. However, for this work YOLOv4 Tiny is used, which applies a combination of stride and pooling layers.

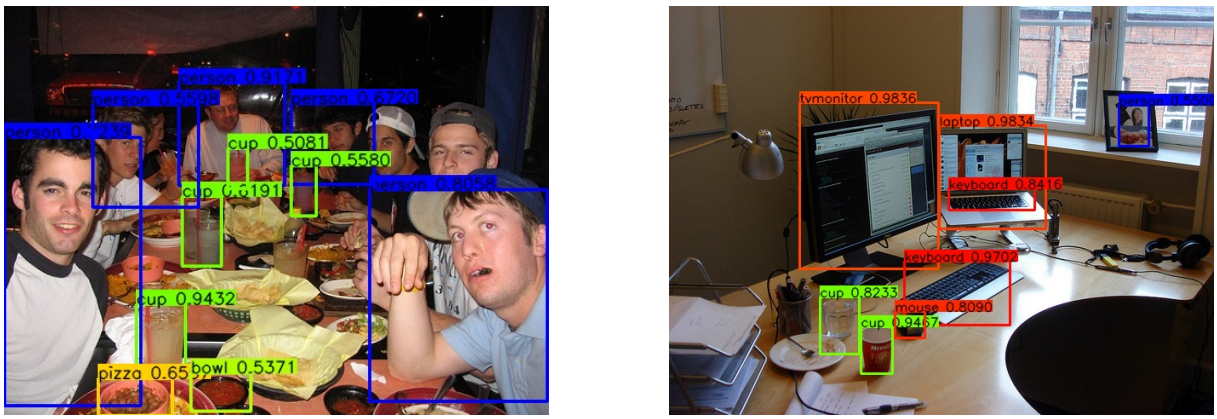
2.1.3 Performance metrics

Section 2.1.2 briefly discussed the topic of training. It is the process of tuning the weights, or parameters, inside the network to provide a better performance. These weights are part of the convolutional and FC layers. Here, the term performance will be defined in the realm of CNNs. It is twofold, on the one hand is the performance in terms of accuracy of the network, as discussed in Section 2.1.3. This is contrasted with the complexity required to achieve that accuracy, which is further discussed in Section 2.1.3.

Accuracy

NNs can be applied to a variety of tasks where the accuracy is determined as how well it performs that task. To make that more concrete for the application of YOLOv4 Tiny, object detection and classification, the accuracy is defined by how accurate the estimate for the object is and whether the identified class is correct. This accuracy is summarised in the mean average precision (mAP) metric. After inference of YOLO or a general object classification algorithm, the network provides one or more bounding boxes for the objects it identified and a confidence score of the most likely class for those items. There are a fixed number of classes based on the dataset that the network is trained with. For this work that dataset is common objects in context (COCO) [15], as it was used to train the YOLOv4 Tiny network by Heinsius [12]. The COCO dataset contains a variety of colour pictures describing every-day life scenes. It contains 80 classes, ranging from *vehicle⇒bicycle* to *food⇒broccoli*. An example of the detection and classification capabilities of YOLOv4 Tiny is given in Figure 2.4; it demonstrates the bounding boxes, which also specify the identified class together with the confidence score.

To determine the accuracy of the bounding boxes, the position is compared to a ground-truth—the bounding box that was manually defined by the creators of the dataset—using the intersection over union (IoU) metric, defined as the overlap of the estimation and the ground-truth, divided by the total area of the estimate and the ground-truth together. If the boxes overlap completely, the IoU is 1. To determine a mAP rating for the network, the IoU value is combined



(a) Pascal VOC 2007 **(b)** COCO

Figure 2.4: Examples of the output from YOLOv4 Tiny on images from different datasets.

with the classification. To do so, four classes are defined. Their definitions are given below:

t_p An object should be identified, and
the classification confidence score is higher than the confidence threshold, and
the predicted class matches the ground-truth class, and
the IoU rating is higher than the IoU threshold.

t_n No object should be detected, and
all values are correctly below their respective thresholds.

f_p No object should be detected, but
the IoU is higher than the threshold, and/or
the classification confidence score is higher than the confidence threshold.

f_n An object should be identified, but
the IoU is lower than the threshold, and/or
the classification confidence score is lower than the confidence threshold.

To determine the accuracy of the network, the type of the prediction is determined using the definitions above for the images in the dataset and a specific object class, e.g. the *vehicle⇒bicycle*. To determine the average precision (AP), the precision (p) is plotted against the recall (r), as given by Equations (2.5) and (2.6) respectively, and the area under this plot is found for recall values in the range [0,1]. To find the mAP, the full description for which is given in Equation (2.7), the average is found for the AP value of all K different object classes in the dataset. This metric determines the accuracy and can be improved by further training the network, e.g. tweaking the weights in the neurons such that the resulting mAP score improves. With a mAP of 100%, the estimation is perfect.

$$p = \frac{t_p}{t_p + f_p} \tag{2.5}$$

$$r = \frac{t_p}{t_p + f_n} \tag{2.6}$$

$$mAP = \frac{1}{K} \sum_{i=1}^{K} \left(\int_0^1 p_i(r) dr \right) \tag{2.7}$$

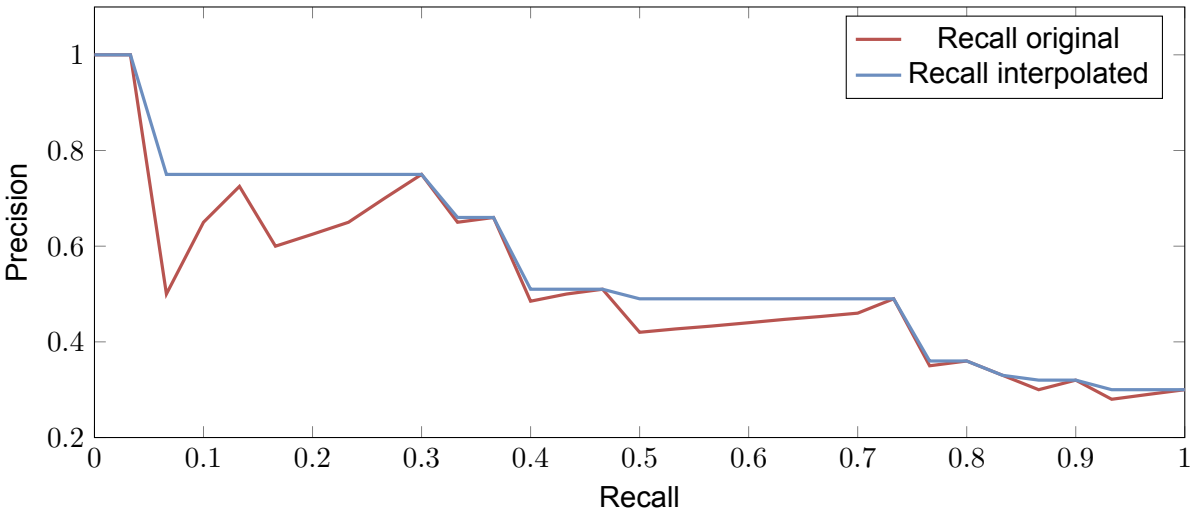


Figure 2.5: A precision recall plot, values derived from [12].

Complexity

A neural network performs a fixed number of computations based on the dimensions of the layers and their constitution. Within the fixed layers with fixed dimensions, the weights of the neurons can still vary, which help determine the mAP. However, for the same computational cost, a neural network may have varying accuracy. During computations, activation functions like ReLU may introduce zeros into the data, which allows for optimisations, as further computations with a zero may be skipped. Without these kind of optimisations, any value will be used regardless in subsequent layers. Hence, each inference of a neural network will require the same number of computations. This number of computations is often expressed in FLOPS, or more generally (also including fixed point operations) GOPS.

For CNNs, the dominant type of computations are the MACCs, as was found by Heinsius[12, Sec. 5.2] who reports 99.67% of the inference time of YOLOv4 Tiny was spend on the *CONV_2D* kernel, so the MACC operations. As such, these operations are often used as benchmark for the computational complexity of a CNN, often expressed as giga multiply-accumulates (GMACC) or GMACC/s. Since the MACC multiplies *and* accumulates, it counts as 2 operations.

2.2 Dimensionality

There are benefits and disadvantages to the CNN computations. The computations are highly regular, with no conditions during operation; hence it is known in advance what data will be used. The downside is that the computations have many dimensions, complicating the scheduling of the computations to efficiently use the available hardware.

To make a schedule it is important to understand the dimensions of the computations. An overview of the different terms is given in Table 2.1[1]. The input feature map (ifmap) represents the input of the NN (layer) and is convolved with filter kernels to produce an output feature map (ofmap). As the convolution requires multiple products to be accumulated, there are intermediate values, called partial sums (psums), to represent incomplete ofmaps. These terms: ifmap, kernel, psum and ofmap are referred to as datatypes as they constitute the different types of data are handled during computations. The formal representation of a 4D convolution is given

[1]The alternative terms are used throughout the implementation in Catapult and the configuration tool of Section 5.4 and are an artefact from the original schedule and configuration tool.

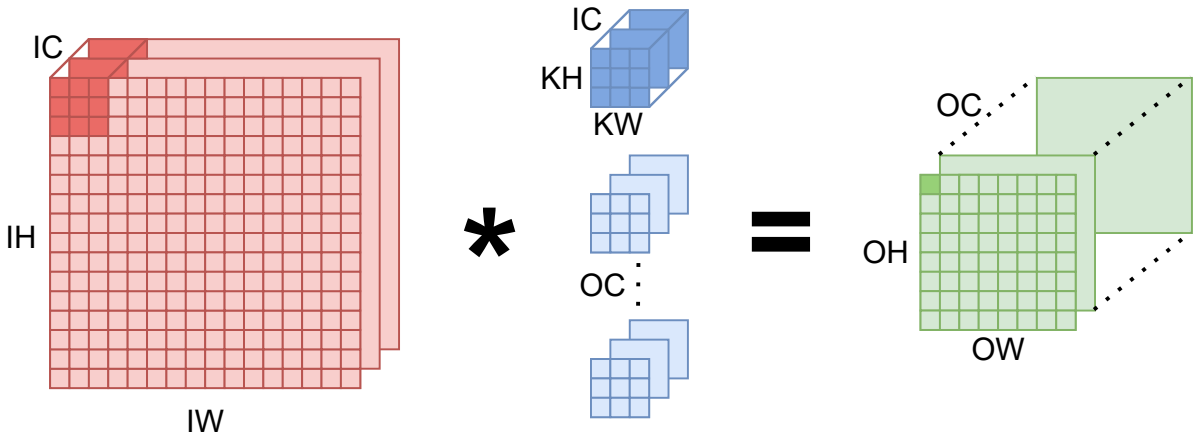


Figure 2.6: Graphical representation of a 3D convolution with a stride of 2, one set of corresponding values is highlighted.

here:

$$\mathbf{O}[oc][oh][ow] = \sum_{ic=0}^{IC-1} \sum_{kw=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathbf{I}[ic][oh \cdot S + kh][ow \cdot S + kw] \cdot \mathbf{K}[oc][ic][kh][kw] \tag{2.8}$$
$$\text{where } 1 \leq oc < OC, 1 \leq oh < OH \text{ and } 1 \leq ow < OW$$

To support the formal representation, the terms are visualised in Figure 2.6, which presents a convolution with a stride (S) of 2. The ifmap and ofmap have 3 dimensions: the width, height and a number of channels[2]. The kernel also has a width and height dimension and a number of channels that must be equal to IC, but there are also output channel (OC) filters to create OC output channels for the ofmap, thus the filter has a total of 4 dimensions. This makes the convolution 4D over dimensions input height (IH), input width (IW), IC and OC. The number of computations required to compute 1 ofmap value is given by:

$$N_{of} = KW \cdot KH \cdot IC \tag{2.9}$$

For the convolutional layer, the only means to reduce the dimensions of the ifmap is using stride. The stride reduces the computations in 2 dimensions. This results in a square factor reduction for non-unit stride. The relation between the ofmap dimensions and the ifmap dimensions can thus be expressed by:

$$\frac{IH \cdot IW}{S^2} = OH \cdot OW \tag{2.10}$$

Given Equation (2.9) as the computations for 1 ofmap position, the number of computations required for 1 layer can be expressed as N_{lay}. With Equation (2.10) this can be expressed by means of the output dimensions:

$$N_{lay} = IH \cdot IW \cdot KH \cdot KW \cdot IC \cdot OC / S^2 = OH \cdot OW \cdot KH \cdot KW \cdot IC \cdot OC \tag{2.11}$$

[2]The input for a CNN is often an RGB-image which has one channel for each colour component, so the ifmap usually has 3 input channels (ICs).

Table 2.1: An overview of the different CNN dimensions, their abbreviations and extreme values for the YOLOv4 Tiny algorithm[25].

Abbrev.	Alt. name	Full name	Min. value	Max. value
S	-	Stride	1	2
B	N	Batch size	1	1
IW	-	Ifmap width	13	416
IH	-	Ifmap height	13	416
KW	S	Kernel width	1	3
KH	R	Kernel height	1	3
IC	C	Input channels	3	512
OC	M	Output channels	32	512
OW	E	Ofmap/psum width	13	416
OH	F	Ofmap/psum height	13	416

2.3 Theoretical reuse

The main advantage of the regularity of CNNs, is the opportunity to reuse data. As values are used multiple times during the computation of the complete ofmap, optimisations can be applied. The theoretical maximum reuse will be derived for the ifmap and kernel datatypes, and reuse hierarchies will be explored.

In Equation (2.9) it was derived that for 1 ofmap value, N_{of} multiplications are required. Since the ifmap has independent channels, 1 ifmap value can be reused $KH \cdot KW$ times. By unrolling the OC dimension of the ofmap, the 4D-convolution can be simplified to 3D convolutions of the same ifmap with OC different filters. Hence, the number of opportunities for reuse, expressed by the reuse factor (RF), for 1 ifmap value is:

$$\text{RF}_{if} = KH \cdot KW \cdot OC \tag{2.12}$$

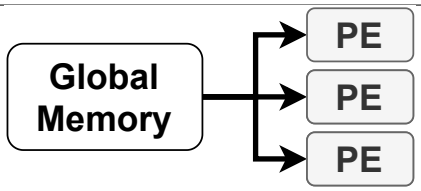
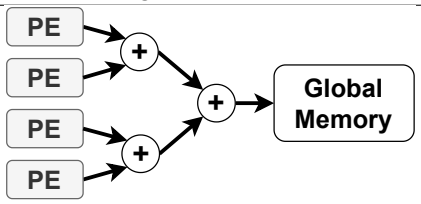
There is a different filter kernel for every OC, and the IC dimension of the kernel matches the IC dimension of the ifmap. This means that 1 kernel value can only be applied to the 2D layer of dimensions $IH \cdot IW$. Assuming that this layer is padded and taking into account the stride, the single kernel value should be multiplied with every ifmap value in this 2D-space making the reuse factor for a kernel value (using Equation (2.10)):

$$\text{RF}_{k} = IH \cdot IW / S^2 = OH \cdot OW \tag{2.13}$$

2.3.1 Topologies

Given these maxima, different topologies can be applied to implement this reuse, they are presented in Table 2.2. The operations are performed on PEs, which represent the atomic computational blocks. For the purpose of CNNs, these PE perform MACC operations. There are two choices for the implementation of reuse: spatial or temporal. For a spatial implementation computations are preformed in parallel, where each PE can be provided with the same data. For instance, RF_{if} parallel executions such that one ifmap value can be broadcast to all computational block to be optimally reused. The outputs of these computations must be accumulated to form 1 ofmap value and stored. By reducing them in a fanin fashion, fewer values need to be stored. As 1 ofmap value is constituted of N_{of} psums, the hardware can be designed to accumulate these prior to storing them, saving transfer bandwidth. The benefit of a spatial implementation is the parallel processing, speeding up a design. The downside is the additional hardware required for this processing.

Table 2.2: Different hardware implementations for spatial reuse, based on [14].

Communication type	Implementation choices	
	Spatial	Temporal
Multicast	 Fanout (e.g. bus, tree)	 Store or forward (e.g. systolic array)
Reduction	 Fanin (e.g. adder tree)	 Reduce and forward (e.g. systolic array)

The alternative is temporal reuse. Here, a value is provided to a (systolic) array where a value can be reused by storing it locally and reusing it over time. There are two options which will be illustrated by providing examples for optimal reuse of an ifmap value. *Store* is when an ifmap can be kept stationary in an element of the array and all $KH \cdot KW \cdot OC$ kernel values can be sent to this element sequentially to compute 1 ofmap in place. Alternatively, there is the *forwarding technique*. Then the ifmap can be sent to the first element of the array, be used to compute a psum with one of the $KH \cdot KW \cdot OC$ kernel values and then forwarded to the next element in the array to be used in a calculation with an other kernel value. The ifmap is optimally reused when the array is RF_{if} elements wide. In both cases, communication bandwidth to the (systolic) array is reduced to the minimum as the ifmap value is only sent once. In the reduction row of Table 2.2, the systolic array is also implemented for reducing the psums and writing back to memory. An example of such an implementation is the situation where the ifmaps are kept stationary, kernel weights can be sent through the array and the resulting psum can be forwarded too, if the array has the appropriate dimensions, the output is the complete ofmap as the only value to be stored back to memory. Temporal reuse provides the benefit of reuse, with the downsides that local storage is required and computations are sequential resulting in longer execution times.

To find a trade-off between resource-heavy spatial reuse and slow temporal reuse, a combination of the two implementations can be constructed that fits the resource and throughput requirements.

2.4 YOLO Version 4 Tiny

The first version of YOLO was released in 2016 for the Darknet framework and delivered real-time performance. Computational complexity is measured in FPS, of which it delivers 45FPS on unspecified hardware. Nevertheless, it provided a mAP of 63.4 on the Pascal VOC 2007 dataset[3] [19]. The design was incrementally improved. Up until version 3 this was done by the original authors. Afterwards, development was taken over and multiple different versions were created. In 2020 version 4 was released which delivers 43.5% mAP on the COCO dataset with 59.56 GFLOPS [4]. To run on an embedded device, 11 GFLOPS is still a large number of computations, so an alternative version YOLOv4 Tiny was designed. It performs worse than

[3]The Pascal VOC 2007 dataset contains 9,963 annotated images containing 24,640 annotated objects out of 20 classes. An example image is given in figure 2.4.

Table 2.3: Layer specifications of YOLOv4 Tiny [25].

Layer	Ifmap dimensions	Total ifmaps	Kernel dimensions	Total kernels	MACCs per ofmap	Total MACCs ($\cdot 10^6$)
0[a]	416x416x3	519168	3x32x3x3	864	27	37.38
1[a]	208x208x32	1384448	32x64x3x3	18432	288	199.36
2	104x104x64	692224	64x64x3x3	36864	576	393.63
3	104x104x64	692224	64x32x3x3	18432	576	98.41
4	104x104x32	346112	32x32x3x3	9216	288	98.41
5[b]	104x104x64	692224	64x64x1x1	4096	64	44.30
6	52x52x128	346112	128x128x3x3	147456	1152	388.56
7	52x52x128	346112	128x64x3x3	73728	1152	97.14
8	52x52x64	173056	64x64x3x3	36864	576	97.14
9[b]	52x52x128	346112	128x128x1x1	16384	128	44.30
10	26x26x256	173056	256x256x3x3	589824	2304	378.54
11	26x26x256	173056	256x128x3x3	294912	2304	94.63
12	26x26x128	86528	128x128x3x3	147456	1152	94.63
13[b]	26x26x256	173056	256x256x1x1	65536	256	44.30
14	13x13x512	86528	512x512x3x3	2359296	4608	358.88
15	13x13x512	86528	512x256x1x1	131072	512	22.15
16	13x13x256	43264	256x512x3x3	1179648	2304	179.44
17[c]	13x13x512	86528	512x255x1x1	130560	512	22.06
18	13x13x256	43264	256x128x1x1	32768	256	5.54
19	26x26x384	259584	384x256x3x3	884736	3456	567.80
20[c]	26x26x256	173056	256x255x1x1	65280	256	44.13
Total	-	**6922240**	-	**6243424**	-	**3310.73**

[a]This layer has a stride of 2, default is a stride of 1.
[b]This layer is followed by a maxpool operation.
[c]This layer has a linear activation function instead of default Leaky ReLU.

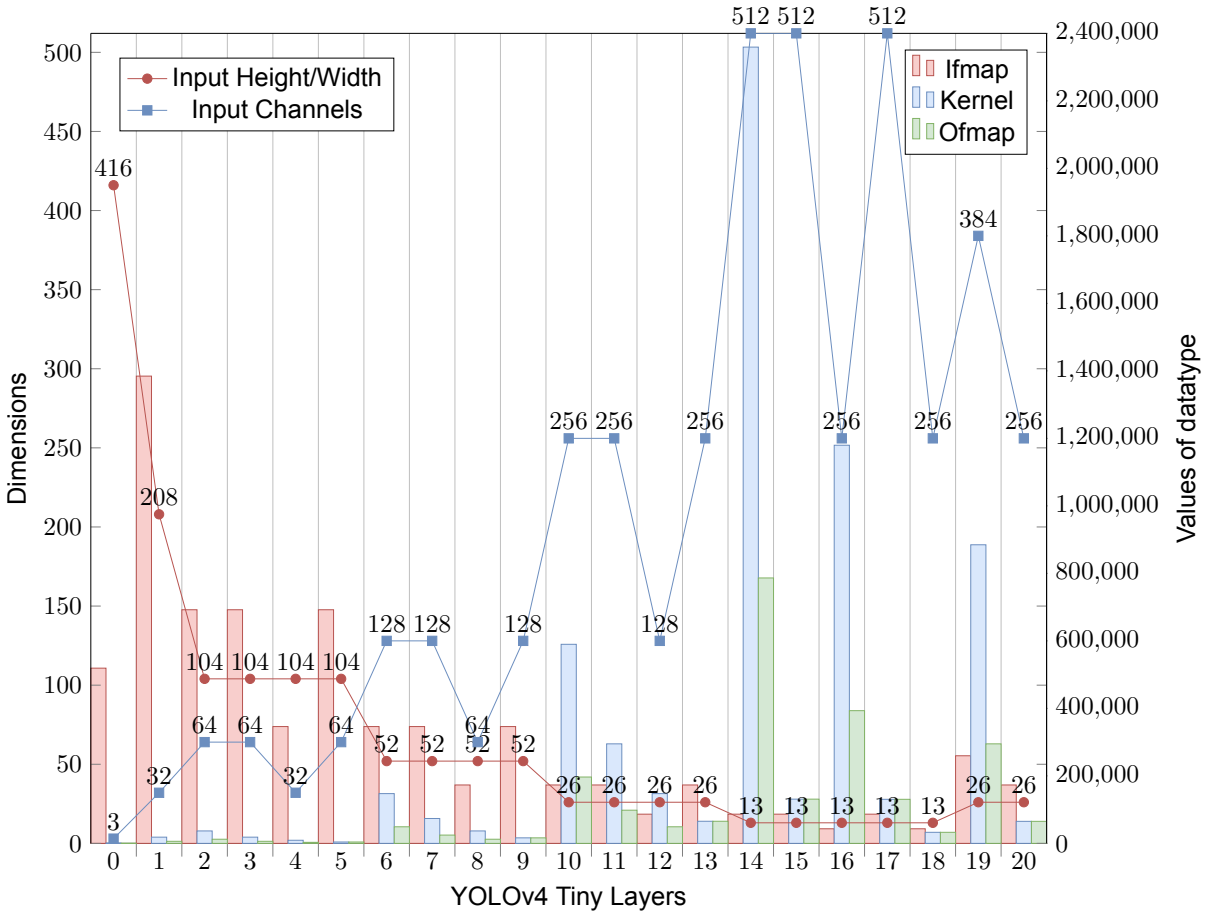


Figure 2.7: The number of values for each layer of YOLOv4 Tiny, per datatype.

the full version 4, at an mAP of 22.0%, but does so using only 6.91FLOPS [25]. The original network uses 16 bit floats for all weights and inputs. The network was ported to the Tensorflow framework by Heinsius to make it compatible with embedded hardware and it uses 8 bit fixed point values. This allows to generalise to 6.91 GOPS for 1 inference of YOLOv4 Tiny. It can be derived from the dimension of the network in Table 2.3 that 1 inference performs 3.31 MACCs, which amounts to 6.62 GOPS, which is 95.8% of the computations.

Next, the relevant characteristics of the YOLOv4 Tiny network are discussed. Its dimensions are visualised and the information from Section 2.3 is applied to this network to provide insights for the remainder of this work. Table 2.3 describes the whole YOLOv4 Tiny network, with the dimensions of each layer's input and the corresponding filter kernels. It also specifies the number of values that are used for every datatype and the number of MACCs required to compute 1 ofmap as well as the total required number of MACCs for the entire layer. These values are visualised in Figure 2.7 and should give insight into the dimensions of the network.

General trends for the network are that the dimensions of ifmaps shrink over time, as stride and pooling layers are applied. The number of channels does grow, which affects the number of ifmaps, but more so the dimensions of the filter kernels. For deeper layers, the number of channels grows, which largely affects the number of kernels. Figure 2.7 demonstrates that after layer 9, the dimensions of the kernels are generally larger than those of the ifmaps. This knowledge can be applied when designing a strategy for loading these datatypes. This strategy might be tailored on a per-layer basis based on the dimensions of the ifmap and kernel.

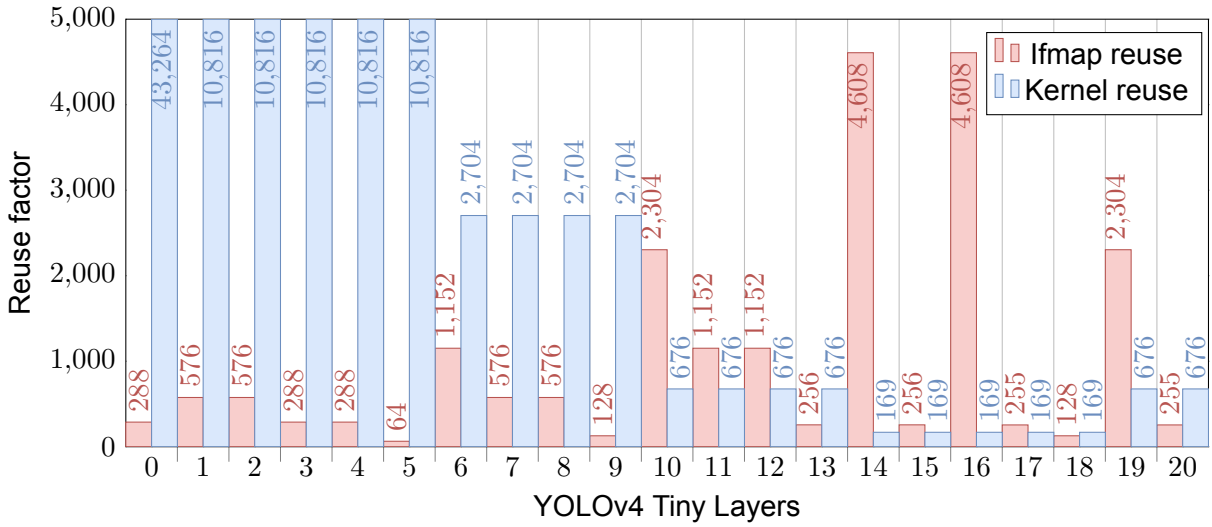


Figure 2.8: Representation of reuse terms for layers of YOLOv4 Tiny.

2.4.1 Practical reuse

Given the bounds for the reuse of the ifmap and kernel, respectively RF_{if} and RF_{k}, it can be found that reuse heavily depends on the layer's characteristics. In layers where the kernel size is 1, the reuse of the ifmaps only depends on OC (which does grow for higher layers). On the other hand, the kernel is dependent on the dimensions of the ifmap, which shrinks for higher layers, limiting the reuse. To indicate the possible reuse over time, these reuse metrics are plotted in Figure 2.8. Again, it can be observed that the first 9 layers mainly favour reuse of ifmaps, whilst the deeper layers allow more reuse of kernels.

Given the characteristics of the layers from Table 2.3 and figure 2.7 and the RFs from Figure 2.8, it can be found layer 5 is most constrained as RF_{k} is low and as the $K = 1$, the number of MACCs is low for this layer. Optimisations should be based around this layer as it might be the bottleneck for computations otherwise.

2.5 Timeloop

The high dimensionality of the computations was discussed in Section 2.2. The number of possible configurations with these dimensions grows very large, especially when combined with different possible hardware layers. To navigate this solution-space, the Timeloop tool [17] has been developed. Timeloop analyses the feasibility of different schedules in the *architecture design space* based on the utilisation of the hardware and energy usage. It is possible to construct a custom hierarchy of memory objects and apply an arbitrary NN by providing a description of the network in a Timeloop format.

The constructed design space will be searched using a search routine, for which there are several options: linear exhaustive, linear pruned[4], random, random pruned[4] or hybrid. Each routine will navigate the space attempting to find mappings which are more optimal for the provided metric. It is possible to have Timeloop optimise for four different metrics: energy, delay, energy-delay product or last-level-accesses (accesses to the outermost memory level). Given the search routine, a number of mappings will be compared on one or more of these heuristics and the most optimal found mapping will be reported. For the exhaustive linear search

[4] A pruned search space removes unnecessary permutations of unit-factors for each possibility at a given dimension that is visited.

Listing 2.1: An example of a Timeloop description of a 1D convolution. **YAML**

```yaml
problem:
  shape:
    name: 1D-Convolution
    dimensions: [ K, O ] # K = kernel size, O = output size
    data-spaces:
    - name: Weights
      projection:
      - [ [K] ]
    - name: Inputs
      projection:
      - [ [K], [O] ] # K+O input values
    - name: Outputs
      projection:
      - [ [O] ]
      read-write: True
  instance:
    K: 3 # Kernels are 3 wide
    O: 16 # 16 Output values
```

all possibilities will be explored (which is very time consuming, for the large search space) and it is guaranteed that the most optimal mapping is found. For the other routines, it cannot always be guaranteed that the best mapping is found.

2.5.1 Usage

In the most simple form, an architecture, the dimensions of the NN, and a description of the operations are provided to Timeloop and it will construct a model of the operations. As output, it indicates the utilisation and energy usage of that schedule. To derive these metrics, Timeloop requires a description like the one provided in Listing 2.1. It is in YAML format and describes the problem given the *shape* of the computations and the size of the *instance*. This description describes a 1D convolution of an ifmap with $\mathtt{O} + \mathtt{K} = 19$ values, with a kernel with $\mathtt{K} = 3$ values. This convolution results in an output map of 16 values. The same syntax can be used to describe higher-order convolutions and complete CNNs. As the dimensions of the network vary from layer to layer, a different description is required for each. However, as the problem is projected on the same hardware, the other descriptions remain the same between layers.

Next to the dimensions of the problem and datatypes, an architectural description like Listing 2.2 is required to ensure the problem fits the hardware. This description is similar to Eyeriss [7], as it also contains 3 memory hierarchy layers; an (off-chip) DRAM, a global buffer (GLB) and a PE, which contains a register file and the MACC operation. The *class* of each object defines the size and speed as well as its energy requirement. The designer can provide an energy estimate or work with the Accelergy tool [27], which is complementary to Timeloop, to derive a power estimate. An accurate estimation here aids Timeloop in estimating the total energy requirement of the schedules it evaluates. The *smartbuffer_RF* class is an RF complemented with the logic to place data in the appropriate location using address generators (AGENs), this increases the energy requirement but is a better approximation of a real architecture. As the goal is to reach the highest throughput, not the lowest energy usage, the exact usage is not relevant. The order of magnitude does bring information, as higher memory hierarchy layers require more energy. If energy usage is low, indicating a lower utilisation of for instance the `MainMemory` (DRAM) or the `GlobalBuffer` (SRAM), reuse is implied. Hence, lower energy usage suggests higher reuse of

Listing 2.2: A Timeloop architecture description of a hierarchy with 3 levels. **YAML**

```yaml
architecture:
  version: 0.2
  subtree: # This subtree specifies all objects in the architecture
  - name: 3level-CNN-accelerator
    local: # Set properties of objects at this level
    - name: MainMemory
      class: DRAM # The class of the memory determines the energy usage
      attributes:
        width: 256
        block-size: 32
        word-bits: 8
    subtree: # This subtree specifies all 'on-chip' objects
    - name: On-chip
      local:
      - name: GlobalBuffer
        class: SRAM
        attributes:
          depth: 8192
          width: 256
          block-size: 32
          word-bits: 8
      subtree: # This subtree specifies all objects in the PE
      - name: PE # Indicate multiple elements using array syntax: [0..X]
        local:
        - name: RegisterFile
          class: regfile
          attributes:
            depth: 64
            width: 8
            block-size: 1
            word-bits: 8
        - name: MACC
          class: intmac # Actual MACC computations are performed here
          attributes:
            datawidth: 8
```

data at the lowest levels.

With these constraints, the design space remains large, even though the hardware architecture is taken into account. To aid the scheduler, the designer can provide additional constraints, primarily to provide a more complete description of the architecture. For instance, to implement the row stationary (RS) dataflow suggested in [7] (see Section 3.2.1), more constraints must be provided to indicate the order of executing the dimensions. There are three different constraints types:

- Bypass: Applies to memories and indicates what datatypes are stored.
- Spatial: Applies to hierarchy layer, allows spatial partitioning, spreading over that loop, e.g. $0 = 1$ indicates that outputs cannot be computed in parallel.
- Temporal: Applies to hierarchy layer, indicates how many of a datatype are stored, e.g. $0 = 1$ does not store outputs at that level. Undefined loops are free to be stored.

Using the *factors* it is possible to indicate that a loop should be spread spatially or in time. Us-

ing the *permutations*, the order of the loops can be indicated, going from first loop on the left to last on the right. Given these constraints it becomes possible to steer the tool in finding a schedule that adheres to the physical hardware, some additional constraints derived from the hardware and finally it is possible to add *mapping_constraints* in the same way as the *architecture_constraints*. These mapping constraints are based on the designers initiative or intuition and can help derive a schedule that is better than what the tool itself might derive based on the search pattern it employs.

Listing 2.3: Timeloop architecture constraints. **YAML**

```yaml
architecture_constraints:
  targets:
  - target: MainMemory
    type: temporal
    factors: O=1 K=1
    permutation: OK # Indicate loop order with left being innermost
  - target: GlobalBuffer
    type: bypass # Specifies what datatypes are stored at this level
    bypass: [Weights] # Don't store weights (they fit in the RegisterFile)
    keep: [Inputs, Outputs] # Do store ifmaps and ofmaps
  - target: RegisterFile
    type: temporal # Indicate how many loops are stored at this level
    factors: O=1 K=3 # Use complete kernel before moving to next ofmap
    permutation: KO
```

3. ACCELERATOR ARCHITECTURES

To determine how well an accelerator performs for speeding up the operation of a CNN. The following lists highlights all relevant metrics according to [24]:

1. Accuracy of the CNN, measured in mAP (see Chapter 2).
2. Throughput of the accelerator, measured in GMACC/s, GOPS or FPS.
3. Latency of the accelerator, measured as the delay from input to output in clock cycles.
4. Energy and power consumption, measured using the average power consumption or expressed as operations per energy unit: GMACC/J.
5. Hardware costs, measured in monetary cost, e.g. $ or €.
6. Flexibility and scalability, measured as how the performance varies when the available resources vary.

Since the goal is to improve the accelerator, metrics 2 and 3 are most important, which can be summarised to overall performance of the accelerator. However, metric 1—accuracy—cannot just be sacrificed to achieve higher performance. It should also be taken into account when comparing accelerators. The accelerator should not be optimised to specific hardware or a specific CNN, to maintain flexibility and scalability, in correspondence to metric 6. Much work has been dedicated to reducing the power consumption of CNN accelerators, but that metric is deemed secondary for this research.

3.1 Definitions

It is concluded that throughput of the design is one of the most important metrics. It is expressed as a quantity of MACCs, operations or frames, per second. Heinsius defines a metric to define the *processing latency* of the CNN accelerator [12]:

$$p_{lat} = \frac{\text{Workload}}{\text{MACC/s} \cdot P} \tag{3.1}$$

Given a batch size of 1 (e.g. processing 1 image at a time), it is inversely related to the throughput. This in turn can be used to express the performance of the accelerator. In the equation, the workload is defined as the number of multiply-accumulates (MACCs) that are required for one inference. P represents the number of processing elements (PEs) (for more details see Section 5.1) processing in parallel. If there are more PEs, and all can be used optimally, the latency is scaled by the number of PEs.

This parameter is largely dependent on the available hardware, as larger FPGA boards can accommodate more PEs. For this reason, the performance should not depend on this parameter and the throughput will be normalised to the number of PEs. As is demonstrated in the work by Heinsius [12], the utilisation of a PE array is not always optimal if the scheduling is not flexible

enough. This may lead to performance differences for varying PE array dimensions or a varying number of PEs. This effect remains when normalising, whilst the scaling effect is removed, providing accurate insight in the performance. This metric—the throughput per PE—is defined as:

$$T_{PE} = \frac{1/p_{lat}}{P} = \frac{\text{MACC/s}}{\text{Workload}} \tag{3.2}$$

The parameters in this equation can be optimised to achieve higher throughput. The first term, MACC/s, should be increased, which demands a higher throughput from individual PEs. This can be achieved by optimising the processing speeds of the PE and providing it with sufficient data to utilise the hardware constantly.

The workload can be reduced, e.g. by reducing/optimising the operations for the current CNN model or developing a new CNN model. For this work, the workload is fixed to 3.31 GMACCs for one inference of YOLOv4 Tiny [25].

3.2 Topologies

In literature, there are many designs for CNN accelerators accelerating various different networks on different hardware platforms, e.g. different FPGA families and dedicated ASICs. They each implement a different hierarchy of the same basic elements: off-chip DRAM, on-chip large global memory (SRAM), on-chip small local memory (scratchpads (spads)/registers) and computational units executing the MACC operations. As larger memories might generally be slower and more power intensive, reuse techniques are implemented to support local reuse of data. These techniques are based on the findings in Section 2.3. The findings from literature are divided into two categories, *data management*, which deals with the distribution of data and the hierarchy described above and *optimisation of computations*, which adjusts the (order of) computations to make them more efficient.

3.2.1 Data management

With data management the goal is to optimise the Accelerator's hierarchy to be most efficient for the computing units or processing elements, thus increasing the MACC/s rating. Heinsius utilises the Eyeriss network [7], but alternative architectures will also be explored, the comparison of which is summarised in Table 3.1. Each architecture benchmarks their performance with the AlexNet NN. This network is not specifically optimised for high throughput performance and accepts 227×227 images as input. It requires 666 MMACCs to classify an image with a mAP of 31.0%, compared to the 3.31 GMACC operations and a mAP of 40.2% for YOLOv4 Tiny [25].

Reuse topologies were explored in Section 2.3. These are also (implicitly) applied by the topologies discussed here. For instance, Eyeriss makes use of fanout data distribution, using busses to deliver data to multiple PEs. VWA [5] also applies the fanout methodology, but uses a tree structure to provide the different blocks with data. The other two architectures, Chain-NN [26] and the Shift architecture [2], make use of a systolic array of PEs to pass on data. Chain-NN passes on both ifmaps and filter kernels, whereas the Shift architecture optimises for passing just the filter kernels.

Eyeriss

Eyeriss [7] is the architecture currently applied in the implementation by Heinsius and it implements a number of novel solutions over other architectures. The authors identify the require-

ment of reusing data to optimise acceleration. To achieve this they optimise data flow using their RS methodology. In contrast to methodologies that optimise for kernel weights, ifmaps or ofmaps exclusively, RS optimises for all data types simultaneously. This is combined with processing convolutions row-by-row (1D), which allows kernel weights to be reused in a PE (temporal reuse). A PE is a structure that performs a MACC operation and contains local memory in the form of spads, one for each datatype, and complementary logic. Within the architecture, all data is initially stored off-chip in DRAM and fetched when required. To enable local reuse, data is cached in a GLB on-chip and sent to the spads to be processed. Communication between GLB and spad occurs using an Y and X bus, to distribute data to a 2D-array of PEs. This allows fanout to occur in two dimensions.

Scaling the hardware is possible, the number of PEs can be adjusted freely and can be used optimally if it matches the dimensions of the CNN algorithm. The main problem when scaling the hardware is the communication. Given the 2D busses that provide point-to-point communication from the GLB to each individual PE, it is not very scalable, as all-to-all communication is costly in hardware.

Eyeriss makes two further optimisations. To reduce the transfers between memories, Eyeriss implements run length compression (RLC) which compresses 'runs' of up to 31 zeros down to a 5-bit number. This compression occurs when data is moving off-chip and to the DRAM. This reduces the bandwidth occupation of transferring the partial sums and thus reduces the power used to fetch data from DRAM. The second improvement is data gating, which is applied in the PEs, and disables computation of a convolution if the ifmap is zero. With zero as an operand, the MACC always adds zero and can thus be skipped. This technique of data-gating is efficient for NNs that apply ReLU-like activation functions, where (many) zeros are introduced in the data. However, as YOLOv4 Tiny uses leaky-ReLU no zeros are introduced by the activation function reducing the effectiveness of this technique. Both improvements aid to reduce the power consumption of the architecture, but do not provide gains in processing speed.

Row stationary dataflow— The RS dataflow is unique to the Eyeriss architectures and aims to reduce data movements, so it is well suited to alleviate the demands on the NoC by reducing the data it needs to transfer, by optimising the reuse at the spad level, so it will be discussed in more depth. "*The RS-dataflow minimizes data movement for all data types (ifmap, filter, and psums/ofmap) simultaneously and takes the energy costs at different levels of the memory hierarchy into account.*"[7] To do so, the 4D convolution are divided into 1D convolutions, where 1 row of ifmaps and 1 row of kernels are convolved to produce 1 row of psums. Each 1D convolution is executed on 1 PE, which allows to keep values constant in the PE. To create an

Table 3.1: Comparison of performance of different data management architectures, all support the AlexNet network with mAP = 31.0%.

	Eyeriss v1 [7]	**Eyeriss v2** [6]	**Chain-NN** [26]	**Shift architecture** [2]	**VWA** [5]
Batch size	4	1	4	1	1
# PEs	168	192	576	96	168
Peak throughput (normalised) [GOPS]	84.0 (0.5)	153.6 (0.8)	806.4 (1.4)	76 (0.79)	168 (1.0)
Latency [ms]	115.3	28.8	353.17	436.4	9.18
Frequency [MHz]	200	200	(up-to) 700	400	500
Power [mW]	278	159.5	567.5	254	154.98
Hardware (CMOS)	65nm	65nm	28nm	65nm	40nm
On-chip storage [kB]	181.5	246	352	43	191

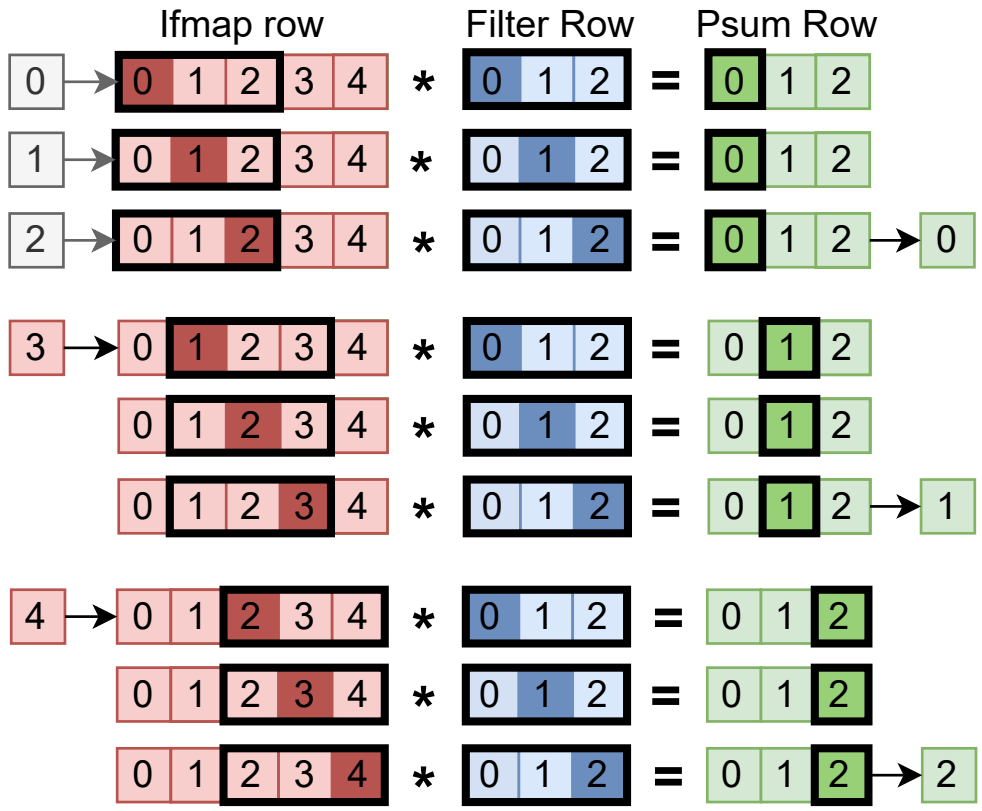


Figure 3.1: The RS dataflow, where weights are kept stationary. Highlighted values are used for the computation, $K = 3$ so three values are accumulated for 1 ofmap. Values inside the black boxes are stored in memory of the PE. Data on the edges shows when it can be loaded/stored.

ofmap, the psums from the different 1D convolutions must be accumulated[1]. Either the ifmaps or the kernel values can remain stationary in the PE, whilst the other can be multicast (see Section 2.3) to different PEs.

Figure 3.1 visualises this dataflow, with a small row of ifmaps ($IW = 5$) and a small kernel ($K = 3$). Weights are kept stationary as new ifmaps are streamed in. The labels on the left indicate when ifmaps can be loaded into the memory indicated by the black box. Values 1-3 can be pre-loaded or could be streamed in directly, whilst new values 3 and 4 are loaded when processing of the first and second ofmap are completed.

Eyeriss v2

Eyeriss v2 [6] is an improvement on the original Eyeriss [7] architecture by the same authors. This architecture is specifically designed to be efficient for compact CNNs. YOLO with its small convolution kernels was designed to be compact and thus can benefit from this structure. It is 42.5 times faster and 11.3 times more power efficient than the original Eyeriss for (sparse) AlexNet, even though it has a batch size of 1 compared to Eyeriss' batch size of 4 [6].

The main improvement over the original Eyeriss architecture is the implementation of an *hierarchical mesh* structure. This structure is implemented between the external DRAM and the on-chip GLB. The available PEs are divided in *local clusters* where each has their own GLB. The mesh network connects all the local clusters. This implementation makes it more scalable

[1] Eyeriss achieves this using a local network (LN), which allows communication between PE that can be used to forward psums such that they can be accumulated.

as there is no all-to-all communication for *all* PEs, but only within a local cluster. This means that adding more PEs does not add much additional routing hardware, as it will be within a new local cluster. The only additional overhead is the new connection of the hierarchical mesh to the new cluster. Given the Zynq 7020 hardware with limited resources, it is unlikely that the accelerator will have too many PEs such that the routing hardware becomes too large.

Besides the improvements in the data management architecture, Eyeriss v2 also implements optimisations for the processing of sparse networks. It compresses data using compressed sparse column (CSC), which compresses successive zeros in a stream of ifmaps *and* kernels to 5 bits, allowing it to accumulate up to 31 successive zeros. To complement this compressed data, the structure of a PE is updated to store both the value and address of the ifmaps and kernels. If either one of these data types is zero, the computation can be skipped and another value can be fetched from the memory in its place.

Chain-NN

The Chain-NN architecture [26] also makes use of PEs to processes the MACC operations. In contrast to the Eyeriss architectures however, a PE in Chain only has a single register to store the intermediate values of the ifmap and psum, but keeps a larger local storage, *kMemory*, to store filter kernels and distribute them in a fanout pattern. New ifmaps and psums are streamed in using the PEs themselves, using the store-and-forward principle. They are arranged in a chain where the input ifmap and calculated psum are passed along the chain of elements, this allows for reuse of the ifmap and accumulation of the psum. The PEs are used optimally if the number of elements is a multiple of the kernel size squared (K^2). This means scaling is possible, but utilisation might be lower for configurations that are not a multiple of K^2. As data is passed along the chain, there is no additional overhead for the transfer of data, other than passing it to the next PE.

As can be observed from Table 3.1, the latency for this architecture is significantly higher than other architectures, as data has to pass each PE in the chain (and Chain-NN also has by far the largest number of PEs) before arriving at the output. This work is a long pipeline, instead on a wide parallel array, thus resulting in a large latency. This is contrasted by the high normalised throughput.

Shift architecture

The architecture in [2] introduces a schedule to process convolutions on PEs. They propose to utilise $IH \times IW$ PEs to perform the required $IH \times IW \times IC \times OC$ computations. The advantages of such a schedule is that kernel weights can be passed on from PE to PE, using the store-and-forward principle. This lowers the bandwidth requirements of the NoC and reduces local buffering, leading to just 43kB of on-chip storage. The utilisation is kept high by reusing PEs, which results in a utilisation factor of $\frac{P}{P+2}$, where P is the number of PEs. In contrast to Chain-NN, the systolic array only passes the kernel weights and not the partial sums. These are accumulated locally in one PE.

The downside to this approach is the high latency, as it takes several clock cycles to process one convolutional frame (for VGGNet): $3 \times (P+2) \times IC \times OC \times \frac{OH \times OW}{P}$ clock cycles.

By increasing the number of PEs, the utilisation of each PE goes up and more PEs are used leading to higher parallel processing and higher reuse of the filter kernels. On the other hand, more accesses to the ifmaps are required, leading to more bandwidth usage for this data type. Thus, it is possible to scale this architecture, without introducing much overhead. Higher bandwidth is required for the ifmaps, but kernels are provided using a general network which is used more optimally if there are more PEs.

Table 3.2: Comparison of performance of different efficient computational networks.

	Fused Layer [1, 32]	**Winograd** [3]
Network (mAP)	AlexNet (31.0)	YOLOv2 (48.1)
Batch size	1	1
# PEs	448	153
Peak throughput (normalised) [GOPS]	61.62 (0.14)	281 (1.84)
Latency [ms]	21.61	124
Frequency [MHz]	100	125
Power [mW]	18610	2700
Hardware (FPGA)	Virtex-7 XC7V690T	PYNQ-z2 XC7Z020
On-chip storage [kB]	362 (36,864 in [32])	880

VWA

The final architecture, VWA [5], employs a structure similar to Eyeriss v2. It groups arrays of 7×3 PEs into *PE-blocks*. There are global blocks of on-chip SRAM that cache data for the different data types, which can multicast to the PE-blocks. Within the PE-block, weights are stored in a local SRAM and passed to all PEs in a column (fanout). Ifmaps are stored in 3 local SRAM blocks, which provide individual data for each element in a row. The resulting psums are accumulated horizontally (fanin), where data can be transferred between rows. With this structure there is no possibility to address individual PEs, so no all-to-all communication is required which means that this architecture is more scalable. More PE-blocks can be added which are supplied with data from the global SRAM blocks.

This work has the lowest latency, which can be accredited to the fan structure with a high degree of parallel processing. It also has a relatively high clock speed, which makes it outperform the Eyeriss architectures.

3.2.2 Optimisation of computations

Given a trained CNN architecture, YOLO in this case, the computations that are to be performed are set. They are divided in a number of layers and provide a defined accuracy, expressed in mAP. New CNN networks are designed and trained to yield higher accuracy with fewer computations. Nevertheless, there are opportunities to simplify the workload (in Equation (3.2)) for a given CNN network. This section explores some opportunities identified in literature, the performance of which is summarised in Table 3.2. What stands out is that unlike Section 3.2.1, the three implementations ([32] and [1] are largely similar) make use of different CNN networks, making the comparison less fair.

Fused layer

Most algorithms process CNNs layer by layer, as the input for one layer is the output of the one before. As local storage is limited, this often results in ofmaps to be stored off-chip before initiating a new layer. Storing and loading these maps takes time and consumes energy as the external DRAM is addressed. This work fuses the processing of layers to significantly reduce the amount of off-chip traffic[1]. As the computations in a CNN network are very regular, it is known beforehand what input pixels are required to compute specific output pixels. This concept can be extended over multiple layers, such that one pixel a few layers down can be computed directly from the input map, by directly continuing processing on the intermediate results, instead of storing them. This alters the order in which computations take place, which

alleviates the bandwidth of the off-chip memory, but reduces flexibility and reuse of filter kernels, as several layers have to be loaded to compute one pixel and then reloaded to compute another.

The algorithm utilises [32] as a basis for the CNN computations. The performance in Table 3.2 is largely based on the data from [32], under the assumption that the fused layer algorithm performs equal or better. The architecture does not make use of PEs, but implements processing engines which have a multiplier for each input, followed by an adder tree to accumulate the products. The optimal dimensions for these processing engines, according to the paper, are 7 inputs and in total 64 engines, resulting in an equivalent of 448 PEs.

Winograd algorithm

The Winograd algorithm proposed in [3] reduces the number of multiplications and instead requires more additions to compute the convolutions in a CNN. As adders are more efficient in hardware than multipliers, this could aid to reduce the utilisation of hardware. This comes at the cost of some mAP accuracy loss, 8.32% for YOLOv2, when also adjusting the bit-width representation from 32 bit floats to 8 bit fixed-point [3]. Here the convolution of input values i and filter kernel values k, is converted to an addition of multiple p (psum) terms:

$$\begin{bmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} p_0 + p_1 + p_2 \\ p_1 - p_2 - p_3 \end{bmatrix} = \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{3.3}$$

Where:

$$\begin{aligned} p_0 &= (i_0 - i_2)k_0 \\ p_1 &= (i_1 + i_2)\frac{k_0 + k_1 + k_2}{2} \\ p_2 &= (i_2 - i_1)\frac{k_0 - k_1 + k_2}{2} \\ p_3 &= (i_1 - i_3)k_2 \end{aligned}$$

Three p terms need to be added (or subtracted), on top of the computations to derive the p terms. The latter consists of an addition of two input terms, before multiplying with a (composition of) kernel term(s). The filter terms k_0 and k_2 are straightforward to use, but the other two are composites of multiple kernel terms. Several options can be considered to work with these kernels. The composite terms could be pre-computed to reduce online overhead, but 4 terms would need to be loaded on chip, instead of 3 for the regular computation. Alternatively, just the $k_1 + k_2$ term could be pre-computed and then reused to compute the composite kernel terms for p_1 and p_2 respectively. This would require an addition and a bit shift for each term.

The reduction of the number of multiplications for 1 channel of the network is considerable, from $OH \cdot OW \cdot KH \cdot KW$ for regular convolution to $(OH + K - 1)(OW + K - 1)$ [3, eq. 4] for the Winograd convolution. For the first layer of YOLOv4 Tiny, where $OH = OW = 208$ and $KH = KW = 3$, the Winograd convolution requires 11.3% fewer multiplications when compared to a normal convolution.

What stands out from Table 3.2 is that the implementation in [3] utilises 880kB of on-chip storage. This is likely dependent on the specific implementation. On the other hand, the peak throughput normalised to the number of PEs is high, with over 1GOPS per PE.

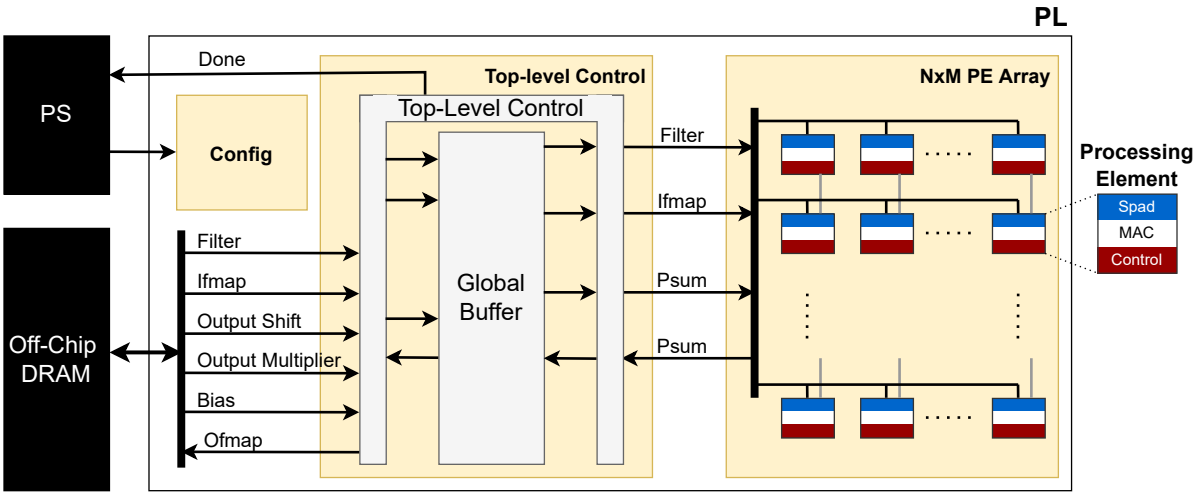


Figure 3.2: Overview of entire accelerator by Heinsius [12].

3.3 Implementations

The Zedboard, or more generally the Zynq 7020 FPGA is used more often for CNN inference [12, 3, 31, 9], the performance of these other works can be used as a benchmark to compare against the contributions of this work and help define the theoretical upper bound for the performance of the Zedboard. To make a proper comparison, the performance of selected other works utilising the Zynq 7020, is compared in Table 3.3. Two works will be analysed in more detail. First, the work by Heinsius will be discussed, as it provides the basis for this work. Secondly, an implementation using Xilinx' deep learning processing unit (DPU) is discussed as it provides the highest performance and is designed by the developer of the FPGA.

3.3.1 Heinsius

The accelerator by Heinsius implements a complete design utilising both the PS as well as the programmable logic (PL) of the Zedboard. The application runs a version of Tensorflow [16], optimised for microcontrollers without an OS, Tensorflow Lite Micro[8], on the PS. After profiling the inference of YOLOv4 Tiny running on only the PS, Heinsius identified that 99.67% of the computation time is spent on the *CONV_2D* kernel. This is expected for a CNN, and indicates that accelerating this operation brings significant performance gains. Hence, the PL is configured to perform convolutions as they are offloaded from the PS. The PS remains in charge of all other computations required for the CNN; applying the activation functions, processing non-convoluting layers, etc. Communication between the two platforms is performed using AXI-busses and a shared memory. Prior to inference, the PS can store data on the shared memory. During inference, the PL can load data from this memory, process it and store it back. Figure 3.2 describes this entire overview and provides a more detailed description of the PL which is discussed next.

Programmable logic

The PL is subdivided into three parts, *configuration*, *top control* and the actual *PE array*. As the schedule and dimension and thus order of execution varies between layers of the YOLOv4 Tiny, different configurations are loaded onto the PL. The configuration block sets loop boundaries for the top control and the PE array. Inside the PL, there are two local memories that avoid the long delays of fetching from the off-chip DRAM. The top control block contains the GLB that

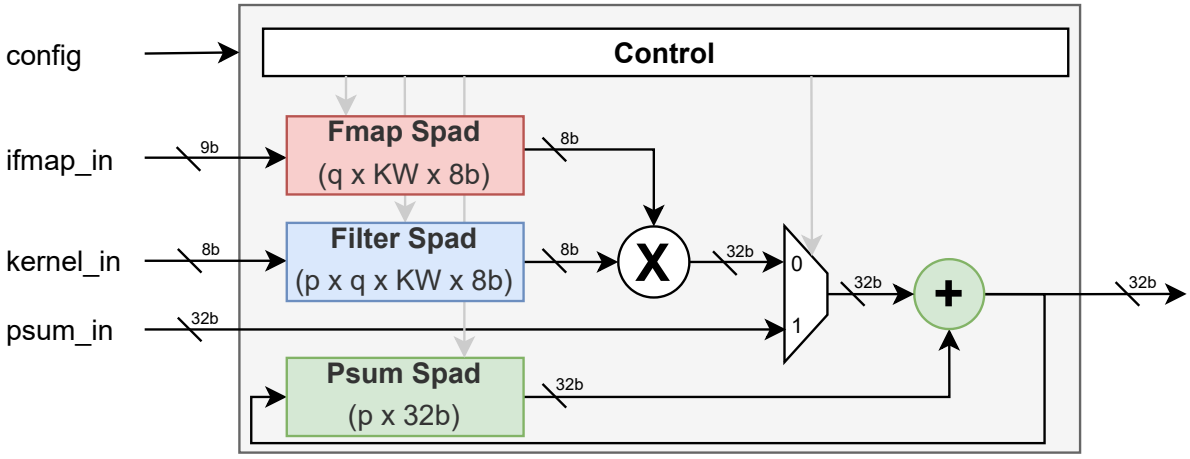


Figure 3.3: The PE implemented in the work by Heinsius, where `p` represents the number of output channels processed by the PE and `q` the number of input channels.

is implemented as SRAM and can store all datatypes. The GLB is filled in the order specified by the configuration, using AGENs, of which there is 1 for each datatype. Data is streamed sequentially into the PE array. The array is built up from individual PEs, which each contain three parts; spads (the most local storage level) to store each of the different data types, the MACC operation and the logic to determine the order of operations (configurable with a signal from the configuration block).

The PE is given in Figure 3.3. It is supplied with data of the different datatypes (ifmap, kernel and psum). As new data is available on the input ports, it is immediately written to the respective spad. The *config* input and *control* block dictate where the data is stored in the spad. The `psum_in` is directly accumulated with the value stored in the spad position, using the adder which is implemented inside the psum spad, this utilises the read and write opportunity of the spad for that cycle (assuming the spad to be simple, this exhausts the read-write opportunity). This method requires an additional cycle to accumulate the new `psum_in`, as it is multiplexed with the product of the multiplier.

The multiplication and accumulation are disjoint, separated by a multiplexer, where the accumulation is performed in the psum spad. By grouping these computations, a consolidated MACC operation can be performed which, when implemented correctly, can be executed by the DSP-blocks on the FPGA. In the current implementation the computation of 1 psum requires 6 cycles.

The NoC is implemented using an X- and Y-bus, to provide the 3x4 array of PEs of data. In contrast to the Eyeriss implementation with 168 PEs and bus-width of 4 elements, the implementation by Heinsius only supports 1 element of a type of the bus [12]. It is identified that the throughput of the NoC in this way might be a bottleneck for the performance, as not enough data is available to perform the computations in the PEs. This might raise the efficiency from the 24% in Heinsius implementation [12] to the optimal maximum of 100%, allowing all PEs to compute continuously.

3.3.2 Xilinx

It must be noted that the throughput for the Xilinx implementation of ResNet-50 is only theoretical, as the Zedboard is not able to supply enough power to support a clock speed of 200MHz for the designed operation [9]. Instead, a clock speed of 90MHz is recommended, bringing the performance down to 103.5GOPS.

Table 3.3: Performance results of CNN accelerators on the Zynq 7020 FPGA, from literature.

	Heinsius [12]	**Winograd** [3]	**Yu** [31]	**Xilinx** [9]	**ZynqNet** [11]	**SCALENet** [20]
Neural network	YOLOv4 Tiny	YOLOv2	YOLOv3 Tiny	ResNet-50	ZynqNet	AlexNet
DSPs	19	153	160	164	739	?
Peak throughput [GOPS]	0.12	281	10.45	230	0.27	2.05
Latency [ms]	59091.4	124	532	16.5	1955	390
Frequency [MHz]	125	125	100	200	200	100
Power [W]	2.32	2.70	3.36	?	7.8	2.85

The Xilinx documentation for the *DPUCZDX8G* DPU [29] specifies several tiers of convolution cores that incrementally use more hardware to provide more performance. The largest size that fits the hardware dimensions of the Zedboard is the *B1152* tier, which theoretically can provide a throughput of 1150 operations per clock cycle. According to the documentation, the theoretical maximum output for the Zedboard is 230GOPS [29], which is in line with the figures found in [9]. The B1152 tier accelerator does not utilise all the available DSP-blocks in the hardware, 97 remain unused.

3.3.3 Summary

The performance of the works that have been described is summarised in Table 3.3. Furthermore it should be noted that all works are designed using the Vivado HLS tool, with the exception of the Xilinx DPU design, which is a custom IP block by Xilinx. When comparing all works, it can be found that the power utilisation is in the same order of magnitude for all works. However, with the same power budget, some accelerators are able to provide more performance, as (peak) throughput varies largely. It can be found that despite using a more modern network (YOLOv4 Tiny), the throughput by Heinsius is lower than other works. This demonstrates that the capabilities of the hardware are underutilised by the accelerator.

As not all works list the number of processing elements, or similar structures, the number of DSP blocks is listed. What stands out is the low utilisation for the work by Heinsius compared to all others and especially the ZynqNet implementation (which uses by far the most blocks). It is likely that the number of DSP blocks is indicative of the number of PEs in the design, which indicates a far lower degree of parallel processing in the work by Heinsius. This leads to a lower throughput when compared to other works and in turn the latency to far exceed that of other works. Given that all works are developed using HLS, it is unlikely that HLS causes the large difference in performance.

The DPU design by Xilinx demonstrates that a continuous throughput of 103.5GOPS can be achieved on the Zedboard. This theoretical throughput may serve as an upper bound for the performance of the CNN accelerator. Given the required GOPS per inference of YOLOv4 Tiny of 6.910, roughly 15 images could be processed per second.

A note must be made that the Winograd work reports a maximum throughput of 281GOPS, exceeding the limit specified by Xilinx. This might be thanks to the Winograd algorithm which reduces multiplications to additions which might reduce the workload and this improve throughput.

4. HIGH-LEVEL SYNTHESIS

The hardware accelerator will be implemented using the Catapult HLS tool. To understand the implementation and the optimisations that are applied, this chapter provides an introduction for the tool and highlights relevant aspects.

The premise of HLS is that hardware can be described using higher-level languages, like C, C++, System C or Matlab. With these languages, development of the hardware design should be faster as development with these languages should be more straightforward. Partially so because HDL related practises, like `clock` and `reset` signals, are not required. The HLS tool will convert this high-level description into a hardware description (for most tools in either Verilog or VHDL).

Like hardware description languages, the high-level language should be used to describe hardware and not for programming. An example is given in [22, Ch. 5] on the scheduling of memories. The memory properties should be taken into account, it cannot sustain unlimited reads and writes in a clock cycle. Taking the hardware's limitation into account when describing the hardware will provide significantly better results. The high-level description can be applied to any platform supported by the tool, as the HLS tool can synthesise the design for the appropriate target: some FPGA family or ASIC. The high-level description has limited directives on how to allocate and schedule computations. In Section 4.1.1 directives are introduced that indicate to the tool how computations should be divided spatially. It is the tool however that performs the temporal scheduling of computations to clock cycles, or C-steps as they are known in Catapult.

Besides the development of a design, the verification of this design is also more efficient, as verification can be performed with a high-level language simulator, instead of HDL simulators. In contrast to these HDL simulators, the high-level verification does not have to simulate all clock edges and thus be much faster. When high-level verification succeeds, the HLS tool ensures that the generated HDL corresponds to this description and is functionally equivalent.

These benefits justify the use of HLS during the development of the Accelerator as it should reduce the development time. However, as indicated in Chapter 1, there are concerns if the use of HLS influences the performance of the Accelerator. As higher-level languages are used instead of HDLs, some of the fine-grained control over the hardware might be lost. This will be explored in this chapter.

There are several available HLS tools, including both academic as well as commercial tools. The most common are commercial tools from well-known vendors: Symphony C (Synopsys), HDL Coder (Mathworks), Catapult (Siemens), Intel High Level Synthesis Compiler (Intel/Altera) and Vivado HLS (Xilinx). Each of these tools support roughly the same workflow. However, as the constraints set out in Chapter 1 demand that the Catapult tool is used over other tools, it will be described in more detail. Throughout this work, version 2021_1.1 of Catapult [21] and corresponding documentation [23] is used.

4.1 Catapult

The Catapult tool was originally developed by Mentor graphics and has since been acquired by Siemens EDA. It accepts three languages as input: C, C++ and Catapult C. The functionality of those language is largely supported and there are specific constructs that help in describing the hardware. Section 4.1.1 describe how relevant C++ constructs are applied can be applied to this work. Recursion is on the features that is not supported by the tool. The design flow for Catapult lists the following steps[23]:

1. Writing and Testing the Source Code
2. Analyzing the Algorithm
3. Creating the Hardware Design
4. Performing Timed Simulations
5. Synthesizing the RTL Design

The writing of source code in Catapult will be explored elaborately, after which the tools provided by Catapult to evaluate the design will be discussed. The next step afterwards is to walk through the remaining steps of the Catapult design process, 3-5, to find how the design is brought from a valid high-level description to a synthesised RTL design.

4.1.1 Writing source code

To understand how the Accelerator is designed an implemented in Catapult, some C++ coding concepts and how they are applied when using Catapult HLS must be understood. These concepts are discussed in this section.

Datatypes

As a high-level programming language is used to descirbe hardware, more accurate descriptions of (sub-byte) datatypes are required. Catapult allows the implementation of different number formats, including integers, fixed and floating point numbers with variable bit widths. These datatypes are described using the syntax in Listing 4.1:

Listing 4.1: Formats for specifying different Catapult datatypes. **C++**

```
ac_int<W,S>      // requires #include <ac_int.h>
ac_fixed<W,I,S>  // requires #include <ac_fixed.h>
ac_float<W,I,E>  // requires #include <ac_float.h>
```

Where W: (mantissa) width, S: signedness, I: integer width. Given these definitions, a signed 8-bit fixed point number with 3 integer bits can be expressed as: `ac_fixed<8,3,true>`. Different descriptions can be created for the different types of data in the Accelerator, e.g. ifmap, kernel or psum. For more complex a `Q` and `O` argument can be passed representing the quantization and overflow modes. These are not exploited for this work.

It is possible to use a slice of the bits in these datatypes, which is achieved using the slice functionality in Catapult. It is possible to set and retrieve a slice of bits from one of the datatypes. The two operations are given in Listing 4.2. `lsb` is the position of the least significant bit where the slice will be set or retrieved. When setting a slice, an `ac_int` type is used.

Listing 4.2: Formats for working with data slices in Catapult. **C++**

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

Templates

A hierarchical design in Catapult is described using C++ classes. These can be instantiated and used to create a hierarchy of different building blocks. However, it is not possible to pass arguments to the constructor of a class, which means it cannot be configured individually. Instead, C++ templating functionality can be exploited to make instantiating classes more versatile. Given the datatypes from section 4.1.1 or constant dimensions, it becomes possible to instantiate building blocks given those constraints. An example is given on line 7 of listing 4.3, where the `dType` variable indicates the datatype of the objects that are stored in the circular buffer and `SIZE` the number of objects that can be stored in the buffer. The power of C++ templates can be exploited here to allow the creation of circular buffers for different datatypes, e.g. ifmap, kernel or psum.

Listing 4.3: C++ description of a circular buffer class. **C++**

```
#include <ac_channel.h>
#include <ac_int.h>
#ifndef CATAPULT_INDEPENDENT
#include <mc_scverify.h> // Include for CSS_BLOCK macro
#endif

template<class dType, int SIZE>
class Circular_buffer {
private:
  dType _buf[SIZE]; //Circular buffer
  typedef ac_int<ac::nbits<SIZE>::val,false> pointer; // Create pointer
      with enough bits to point to all objects in buffer
  pointer _read, _write;
  bool _full, _empty, _available[2];

  void reset_ptr(){ // Function to reset entire buffer
    _read = 0;
    _write = 0;
    _full = false;
    _empty = true;
    _available = {0}; //Set all values to 0
  }

public:
  Circular_buffer(){
  #pragma hls_unroll yes
    BUF_RST: for (int i=0; i<SIZE; i++) {
      _buf[i] = 0;
    }
  }

#pragma hls_design interface
#ifndef CATAPULT_INDEPENDENT
  void CCS_BLOCK(run)(
#else
```

```
35   void run(
#endif
           ac_channel<dType> &data_in,
           ac_channel<bool> &read,
           ac_channel<dType> &data_out) {
40     _available[0] = (data_in.size() > 0);
41     _available[1] = (read.size() > 0);

       if (_available[0]){ // Write data to buffer
         // Logic omitted
45       _buf[_write] = data_in.read();
46     }

       if (_available[1]){ // Read data from buffer
         // Logic omitted
50       read.read(); //Read channel to empty it
         data_out.write(_buf[_read]);
       }
}};
```

Pragmas

With custom datatypes and templating, a C++ description can be made. To optimise this description to be converted to hardware, pragmas can be used. These are additional directives on top of existing `#pragma`, to indicate the compiler how the C++ description should be applied in the hardware. Catapult provides different pragmas to indicate different functionality and these can be divided in a number of categories: hierarchy, constraints and directives. The most relevant pragmas are discussed here, all pragmas are described in the Catapult user reference [23, ch. 26].

The hierarchical pragmas indicate how different blocks/classes are interconnected and which class is the top of the design. These are indicated with for instance: `#pragma hls_design interface` for a hierarchical interface and `#pragma hls_design top` for the top. Additionally, properties can be assigned to the hierarchical module, it can for instance be assigned as a CCORE block or black box. Catapult allows allocation of CCORE (Catapult optimised reusable entities) blocks for modules by reusing the synthesised result to shorten the run-time when instantiating these blocks. A block can be indicated as CCORE or black box using `#pragma hls_design ccore` or `#pragma hls_design blackbox` respectively, which can be an addition to the existing `interface` or `top` directive.

Constraints can be set on the performance using a different set of pragmas. These involve unrolling and pipelining of design blocks and loops and largely corresponds to the settings available in the architecture step (section 4.1.2). If these settings are fixed for a given design, they can be set in the C++ description using the following pragmas: `#pragma hls_pipeline_init_interval 4` and `#pragma hls_unroll yes` to set an initiation interval of every 4 clock cycles and allow a loop to be unrolled.

Hierarchy

To construct a complete design it is common practise in Catapult to work with a hierarchy (although it is also possible to implement a design using purely functions). Using pragmas, the top of the design can be designated, and sub-blocks can be labelled with the interface label. To allow blocks to run at the same rate, they should be interconnected using `ac_channel<T>` objects, which serve as a FIFO buffer between the hierarchical blocks. The channel can serve

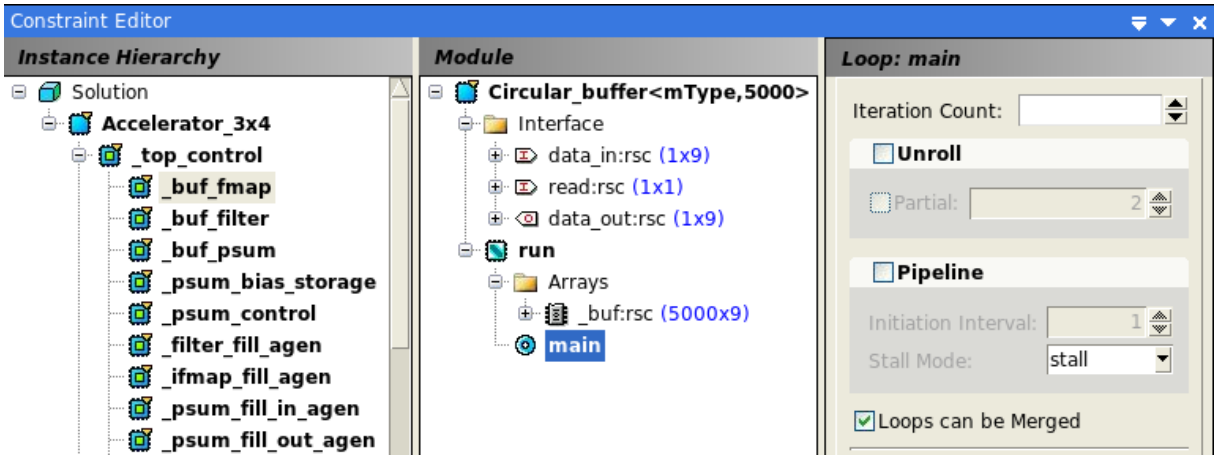


Figure 4.1: The architecture window in Catapult, illustrating *Unroll* and *Pipeline* options for an implementation of a circular buffer, *_buf_fmap*.

to transport different datatypes, including arrays, and are able to provide data at a constant rate. In hardware, data is read from the output of the channel, regardless if data is available and/or valid. In HLS however, a read cannot be performed on an empty channel as no data is available, thus a programming exception is returned. In order to circumvent exceptions when simulating the HLS description, guards are put in place to prevent untimely reads. An example is given on line 40 of listing 4.3, where the size of the `data_in` channel is found, when there are elements in the channel, a read can be performed on line 45, which is otherwise skipped until the next inference of the `run()` function. These guards are removed when the hardware is synthesised as they are then no longer required, providing the best results in hardware.

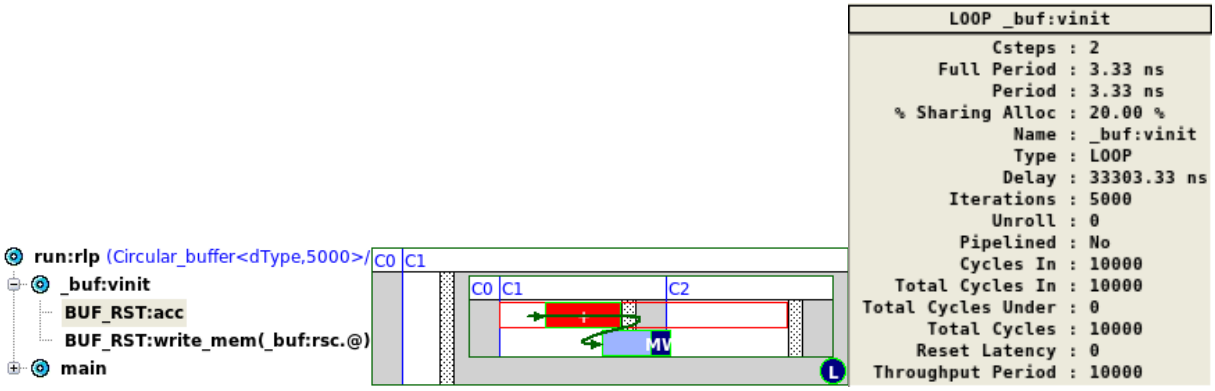
Listing 4.4: C++ description for calling the circular buffer class. **C++**

```
typedef ac_fixed<8,3,true> exampleType;
Circular_buffer<exampleType,16> buf; // Constant size of 16 elements
ac_channel<exampleType> data_in; // Data should be written to this channel
ac_channel<bool> read;
ac_channel<exampleType> data_out; // Read from this channel to retrieve
    data
buf.run(data_in, read, data_out);
```

Any hierarchical block can either contain purely hierarchy *or* purely logic. In hierarchical blocks, the channels are used to interconnect sub-blocks with higher hierarchy layers, or with other sub-blocks. The arguments of the `run()` function are all channels passed by reference, which is mandatory, as it creates the desired connection, instead of instantiating a local channel. Starting from a top-level hierarchical block, the circular buffer example class can be called. For the hierarchy, each class should have a `run()` function, which can have multiple `ac_channel<T>` as an interface. An example is given in listing 4.4, where a custom datatype is created to instantiate a circular buffer `buf`, together with a constant size of 16 elements. Subsequently, channels are created for that custom datatype `exampleType` to connect the buffer to the higher level object from which it is called. By calling the `run()` function, the function body is executed once, this allows it to run at the same interval as the higher hierarchical blocks.

4.1.2 Analysis

This design can be analysed using the tools provided by Catapult. In a graphical user interface (GUI), it is possible to explore the description in the *architecture* window, as in Figure 4.1. This



(a) Schedule viewer

(b) Statistics corresponding to the `_buf:vinit` block.

Figure 4.2: The Schedule window (Gantt chart) in Catapult, showing the performance for the initialisation of a circular buffer.

window shows how architecture constraints are applied to the design: are loops sequential or unrolled for parallel execution? The figure presents the options for unrolling and pipelining the design, by a user settable factor. Catapult will try and generate a design that adheres to these instructions. Like with HDLs, it might not be possible to meet these requirements given a certain clock speed (HDL tools usually lower the clock speed), Catapult report an error. To resolve this either the architecture constraints should be relaxed, or the design must be adjusted e.g. by implementing logic such that it is executable within a single clock cycle.

Schedule viewer

After the architecture setting have been updated for the design, a schedule of the operations can be created. This schedule can be viewed in the form of a Gantt chart, see Figure 4.2. It specifies the operations and how they are scheduled and can be an important tool in identifying bottlenecks in the design and the operations that are slowing down operation. In this example, a circular buffer is initialised in a logical block named *_buf:vinit*. This block is implemented the following way:

Listing 4.3 shows how the circular buffer is initialised by looping through all elements and setting them to 0 on line 26. This loop is unrolled using one of Catapult's pragmas (see Section 4.1.1). This behaviour can also be identified in the Gantt chart, as there are two operations: `acc` for incrementing the loop bound `i` and `write_mem` is the assignment of the `_buf` variable to write to the memory. The arrows in Figure 4.2a indicate the data dependencies, where the write to memory is based on the loop bound `i`, which should first be incremented. The red box around the `acc` operation indicates the freedom in scheduling the operation anywhere in the available time for the loop.

From the statistics in Figure 4.2b it can be found that executing these two operations takes 2 C-steps per iteration. As the `SIZE` of the buffer is set at 5000 elements, the total initialisation takes 10000 cycles as indicated by the *(Total) Cycles In* and *Throughput Period*. Working with the assumption that the buffer is written to before reading, the initialisation value is not relevant. Hence, the initialisation can be updated to:

Listing 4.5: Improved C++ description for initialising a circular buffer. **C++**

```
1 #include <ac_int.h>
static bool initValid = ac::init_array<AC_VAL_DC>(_spad,SIZE);
```

Using the `ac::init_array` function from Catapult's algorithmic C (AC) int library it is possible to initialise the array in 1 execution, in this case initialising all values to `AC_VAL_DC`, don't cares. The function returns a boolean to indicate if the operation was successful. Using this implementation the initialisation occurs during compile time, completely eliminating all delays. Compared to the original initialisation, this removes a significant number of C-steps which improves the throughput of the Accelerator. Using the Schedule viewer, opportunities for optimisations are more easily discovered. To fully analyse a schedule, the meaning of important statistics from Figure 4.2b will be listed here:

Csteps	"*C-steps are roughly equivalent to states in a finite state machine (FSM). If the design has complex conditional statements, several finite state machine state may map to the same C-step.*"[23]
Period	The clock period.
% Sharing Allocation	The percentage of the clock period used for sharing data, see grey area inside the C-steps in Figure 4.2a (default 20%).
Delay	Execution time for 1 inference of the block.
Iterations	# iterations in loop.
Unroll	Unroll factor for the loop (0 is no unrolling).
Pipelined	Is the loop pinelined.
Initiation	Initiation interval of the pipeline.
Stages	The # stages in the pipeline.
Total Cycles In	# of cycles in loop (Iterations + ramp-up).
Total Cycles Under	The # cycles inside this block.
Total Cycles	Total number of *Cycles In* and *Cycles Under*.
Reset Latency	Latency for resetting, dependent on memories in the design.
Throughput Period	The throughput rate of the block.

4.1.3 Building for a target

Finally, the optimised design should be prepared for the hardware. Here, some final configurations must be performed to give a complete description of the hardware. The same design can be built for a variety of targets, based on the available hardware libraries. For this work the Zynq family of FPGAs is targeted, specifically the Zedboard (xc7z020clg484-1). With the library description of this device, Catapult creates a default configuration with a selection of hardware for the different operations and memories. One of the goals is to utilise the DSP-blocks available in the Zedboard. During normal synthesis opportunities to apply DSP-blocks are inferred by the synthesis tool. As the design flow with Catapult is different, Catapult should infer the location of DSP-blocks in the description. Catapult is only able to extract DSP operations for the Xilinx Virtex-u (plus) and Altera Stratix 10 and Arria 10 families [23, Ch. 22]. An alternative is sought for implementing DSP-blocks outside of the automatic design flow, this topic is further discussed in Section 5.1.2.

The architecture step also includes options to configure what hardware is used. It is possible to select individual components for operations (Catapult selects the best area component that meets the timing requirements) and to select the hardware for memories. The circular buffer has the array element *_buf:rsc* for which the memory is set to a Xilinx RAM Block by default, based on the (large) size of the buffer. However, alternate RAM blocks can be selected, or the choice can be made to implement this memory as a register file in favour of performance. After setting these final configurations, register transfer level (RTL) files can be generated.

Steps 4 and 5 from the list (see Section 4.1) are left. Given the RTL files, timed simulations can be performed to verify the performance. These simulations can be performed in the simulation

tool of choice, e.g. Questasim or Modelsim. Simulations can be performed on both a behavioural and RTL level. If the design meets the requirements, it can be synthesised, using an external tool. As the Zedboard is a Xilinx device, Vivado is used to perform this final step. Given the RTL files, this step is straightforward, as Vivado is launched and a TCL script executed to synthesise the RTL design.

5. METHOD

The work by Heinsius serves as the foundation for this work. In Section 3.3.1 the design of the hardware accelerator implemented in the Zedboard's PL is discussed. A number of shortcomings are identified which will be addressed. In this chapter, the accelerator will be described bottom-up. Three main topics will be discussed. At the lowest level, the throughput of the PEs is improved in Section 5.1. For this, the implementation of the PE should be adjusted such that the multiply and accumulate computation are performed in the same place. Next, that place will be implemented as a black-box in Catapult that can be exchanged for a DSP48E1 primitive to ensure that the MACC is executed on the Zedboard's DSP primitives.

Secondly, the structure of the PE array—implementing the individual PE blocks—will be revised in Section 5.2. In the work by Heinsius the array is implemented as a 3x4 array, with two dimensions, as derived from the Eyeriss architecture. For the application specific integrated circuit (ASIC) application of Eyeriss it is logical to take spatial planning into account such that PE are spread over the silicon. Since the array description from Catapult is synthesised and then mapped to the FPGA fabric, the spatial information from Catapult is not relevant, as it is the synthesis tool that will provide a mapping for the hardware. That is why the array will be implemented as a 1D array. This simplifies both scheduling and the NoC providing data to the PEs. The new structure also demands a different schedule, created with Timeloop, to optimise the computations for the new hardware implementation.

The third and final improvement is an update to the NoC to ensure that it is no longer the bottleneck for the computations. The first step is already made, as the PE array now requires just one bus, reducing complexity of the NoC. This is discussed in Section 5.3.

5.1 Throughput of the PE

The very core of the accelerator are the MACC operations, of which billions are required to compute a layer of YOLOv4 Tiny. In the work by Heinsius, the MACC operation requires 6 clock cycles to complete within a PE and is structurally similar to the PE in Eyeriss. However, the PEs in that work achieve a throughput of 1 on their ASIC. An improvement could be expected to approach the optimal throughput of 1 MACC per clock cycle for the Catapult implementation. It requires 6 cycles due to the current implementation. Catapult is not able to schedule or pipeline the operation required for the PE to within 1 cycle. For the multiplications, a XXX pipelined multiplication component is used. Using the pipeline, a product is produced every cycle. Next, the addition needs to be performed which is also requires 1 or more cycles. Finally there is some surrounding operations which account for the last cycles, adding up to 6.

Since there are multiple PEs in the Accelerator, the throughput of the entire accelerator is higher, but by optimising the throughput of the individual PE, the performance of the accelerator increases by a factor equal to the number of PEs. The original design was analysed in Section 3.3.1 and two points of improvement were found for the PE. The *psum in* should be multiplexed with the value from the spad instead of the product of the multiplication to avoid stalling

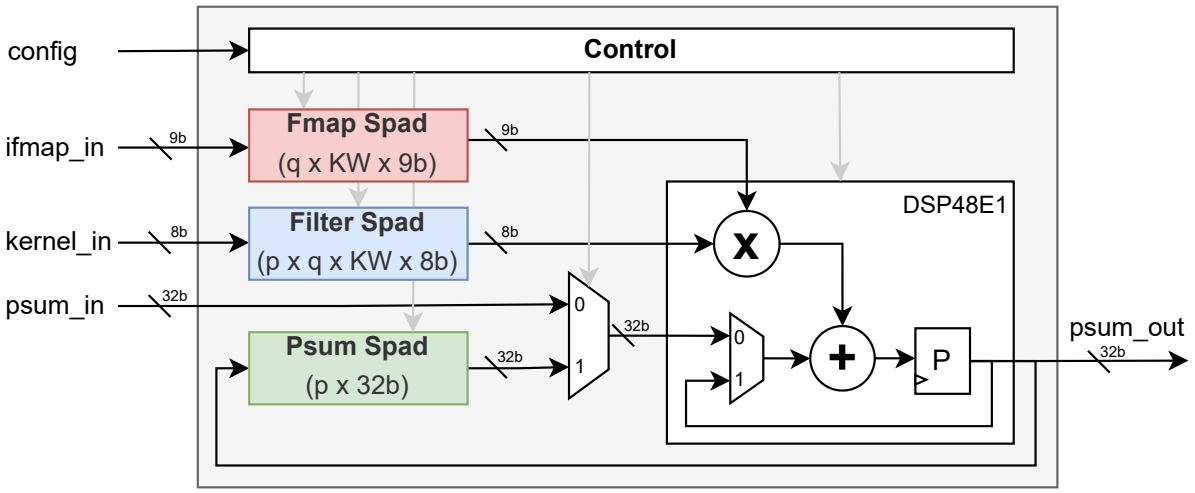


Figure 5.1: Improved PE, where MACC is implemented in a DSP48E1 primitive and *psum_in* is multiplexed with the spad output.

computations when loading new data.Also, the multiply and accumulate operation should be consolidated, such that it can be implemented in a DSP primitive. A topology with these improvements is given in Figure 5.1.

5.1.1 DSP48E1 primitive

The goal is to increase the throughput of the individual PEs. This improvement should be twofold, the computations, the multiply-accumulate, should be performed faster and data should be readily available for these computations. This demands a speedup of the spads, as the ifmap and kernel spads should be able to provide 1 value per cycle. According to [12, Ch. 7], more resources are available as none of the fabric's resource types is fully utilised.

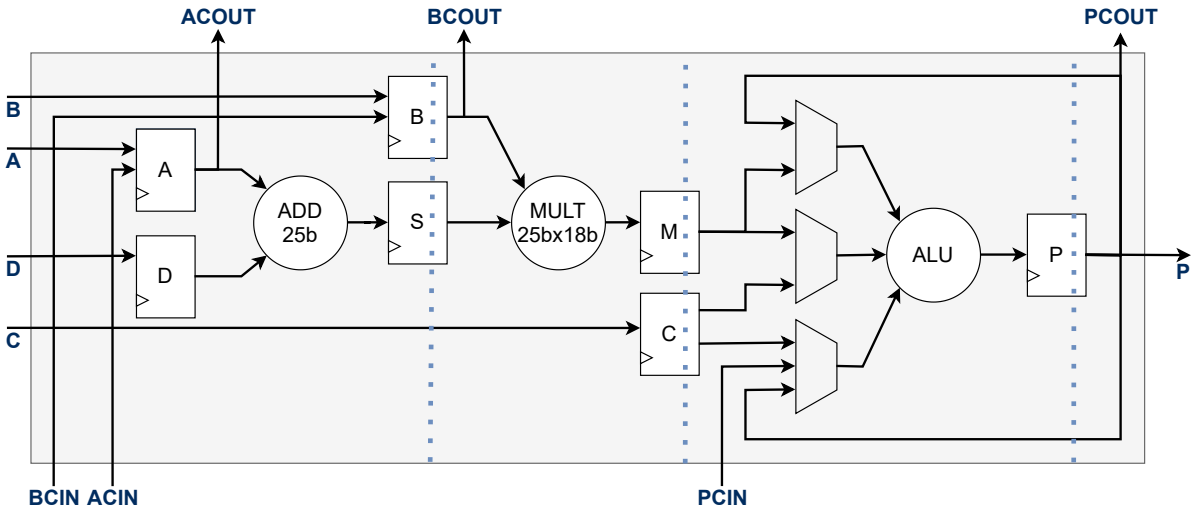


Figure 5.2: A simplified DSP48E1 slice.

In the original design, the DSP-blocks go unused for the MACC computations. DSP-blocks in the FPGA fabric are able to provide custom digital signal processing operations at high efficiency, both in power and speed. The DSPs available on the Zedboard are the DSP48E1 primitive, which are available on all 7 series Xilinx FPGAs. They provide a range of configurable operations, including the MACC [28]. Figure 5.2 represents a simplified overview of the

DSP primitive. It contains a number of internal registers to allow for pipelining operations and subdiving the block into 3 stages: pre-add[1] (A+D), multiplication (B*S) and an ALU operation, for this purpose MACC (M+P) or post-add (M+C).

The in and outputs of this primitive can be divided into two types, the user accessible (A, B, C, D and P) and the ports *only* available within the DSP-column in the FPGA fabric to allow inter-DSP communication (ACIN, BCIN, PCIN, ACOUT, BCOUT and PCOUT). The first can be manually connected, whereas the latter is routed by the synthesis tool through the DSP-column. Using control signals it is possible to select between either of the ports, during run-time. This enables the forwarding of partial sums to the next PE in the column, or the sharing (broad- or multicast) of ifmap and kernel values between PEs, like the store and forward methods from Section 2.3.1.

In general operation, the A and B ports would be used to provide the ifmap and kernel values. Depending on the computation, the resulting product can be accumulated in register P, or the C-port can be utilised to provide offset values as an initial value for the psum. The P value can be passed out of the DSP block using the PCOUT port to an adjacent PE and be used to accumulate along PEs, or directly passed out of the DSP using the regular P-port.

5.1.2 Black-box implementation

Catapult provides the opportunity to implement computations on DSP-blocks using the `DSP_EXTRACTION` directive. However, this directive is only supported for selected families: Xilinx' Virtex-u and Virtex-u plus and Altera's Stratix 10 and Arria 10 [23, Ch. 22]. The intended target for the Accelerator is Xilinx' Zedboard, with a Zynq family FPGA, which is thus not supported. As the DSP-blocks are paramount to realising the intended performance, an alternative is required.

According to [23, Sec. 2.3] it is possible to instantiate custom IP-blocks in a Catapult design. In Vivado an IP-block is available for the DSP48E1 primitive described in Section 5.1.1. To insert such a block, a black-box is instantiated in its place in Catapult. The black-box is created as a dedicated class, which includes a `run()` function which is preceded by the following: `#pragma hls_design interface ccore blackbox`, which indicates that this class should be implemented as a CCORE black-box. The `run()` function itself should provide a C++ description of the operation of the black-box and the special `ac_blackbox()` instances to provide a link to the corresponding HDL file(s) and information on the area and delay of the box. This implemented as in Listing 5.1. The information to specify this black-box in Catapult can be derived from Vivado; the DSP-block is pipelined, with 4 register stages, resulting in a latency of 4 clock cycles. After synthesis, the area of the DSP IP-block is found to be 1 DSP, 3 LUTs, 98 flip-flops (FFs). Catapult uses the LUTs as indication of the area, so the footprint of a DSP-block is very small.

The VHDL file *xbip_dsp48_macro.vhd* can be created using the IP-catalog in Vivado. The IP-block can be configured to implement a number of operations. To provide optimal flexibility when implementing the PE-array, the DSP-blocks should provide the following functions:

- A(CIN)*B(CIN)+P output: P(COUT)
- A(CIN)*B(CIN)+PCIN output: P(COUT)
- A(CIN)*B(CIN)+C output: P(COUT)

For this work individual DSP-blocks are implemented, making that the *CIN* and *COUT* ports might not be used directly. However, there would be opportunities to create large black-boxes where these connections can be exploited instead of using the regular routing resources of the

[1] The pre-adder can be helpful when implementing e.g. symmetrical FIR-filters, or the computations of the Winograd terms from Section 3.2.2, but goes unused for the MACC operation.

Listing 5.1: The MACC black-box class in Catapult. C++

```
template<class mType, class dType, class rType>
class MACC_BB {
  public:
    MACC_BB() {}

#pragma hls_design interface ccore blackbox
    void run(
              //inputs
              mType ifmap, dType kernel, rType psumIn,
              //outputs
              rType &psumOut ) {
        static rType regP; // P register, maintains value
        regP += psumIn;
        regP += ifmap * kernel;
        psumOut = regP;

        ac_blackbox()
            .library("work")
            .entity("MACC")
            .architecture("bhv")
            .vhdl_files("xbip_dsp48_macro.vhd")
            .outputs("psumOut")
            .area(3)    // #LUTs, 98 flip-flops and 1 DSP are also used
            .delay(1)   // Delay is 1 clock cycle
            .latency(4) // Latency is 4 clock cycles
            .clock_name("clk")
            .posedge_clock(true)
            .end();
    }
};
```

fabric. Moreover, it may provide the desired store and/or forward topologies, as implemented in Chain-NN or the Shift architecture from Section 3.2.

5.2 Processing element array

In Section 5.1, the throughput of the individual PE is discussed. With the introduction of the DSP-blocks, the PE can process a MACC operation in 3 clock cycles. By moving the computational load to the DSP-blocks, resources utilised by the PE-array are freed up. The PE-array amounted for $21177/53200 = 39.8\%$ of total utilisation [12, Ch. 7]. This can significantly be reduced by replacing the MACC logic by DSP-blocks, as they only utilise 3 LUTs. In a PE utilised 646 LUTs, this brings a large reduction for the individual PE, and a reduction in LUT utilisation of 36.44% for the PE-array and 18.36% for the entire Accelerator. The PE-array can be further scaled up with the newly available resources as other resources are still readily available.

Moreover, the array is implemented as a 3x4 array, spatially divided in two dimensions which is inspired by the Eyeriss architecture. However, Eyeriss was implemented in ASIC technology where spatial planning is required. The Accelerator in this work will be implemented on an FPGA, where the fabric is fixed and the design will be mapped to this hardware by the synthesis tool. Hence, there is no strict need for a 2D-array. In fact, the DSP blocks and BRAM modules are aligned vertically in the fabric of Zynq devices [28].

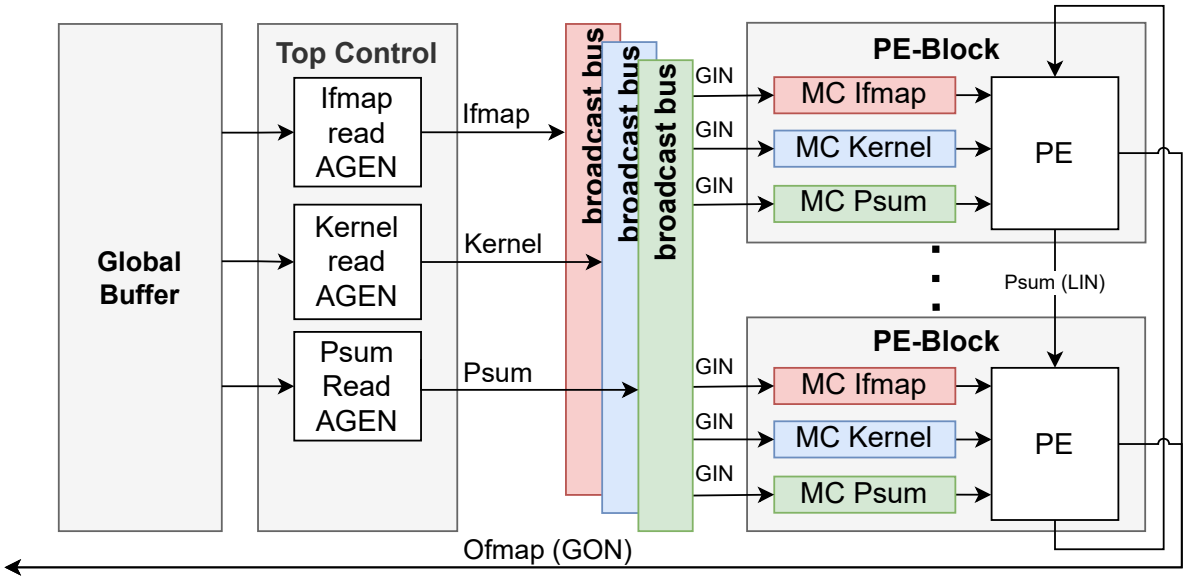


Figure 5.3: An overview of the improved PE-array.

The 2D-array provides the benefit of sharing psums from different 1D convolutions as part of the RS dataflow. By creating 3 rows of PEs, and vertically accumulating, the top PE can output the complete ofmap. It is also possible to share these psums over the 1D array using other means, for instance using the CIN and COUT ports of the DSP primitive, or simply forwarding data along the PEs. The best method will depend on the schedule that will be applied for the calculations. This schedule, with the most optimal utilisation of the array's resources is derived, in Section 5.2.1.

The array is simplified such that the `PE_array` hierarchical block contains P PEs, without an additional hierarchy layer. Figure 5.3 visualises this structure. This omits the 2D hierarchy from Figure 3.2 and broadcasts data from the AGENs in the top control to the PE-blocks. Concerns regarding this structure and the impact of the NoC on the performance of the accelerator are discussed in Section 5.3.

5.2.1 Schedule

As the old Timeloop schedule was made for the 2D-array it is no longer applicable. To make a new schedule, the architecture description for Timeloop must be updated. Additionally, there is an opportunity to optimise the schedule. In the original accelerator, the array was not always optimally utilised; the total utilisation factor for all layers was 66%, which leaves a large margin for improvement. To identify why the original schedule was not 100% efficient, that schedule will be analysed first. Afterwards, the updated schedule will be presented in Section 5.2.1.

Original accelerator

A Timeloop description corresponding to Section 3.3.1 is available, with three hierarchical memory layers: off-chip DRAM, a GLB and the local spads. It is given in Listing 5.2. The architecture contains 12 PEs, in a 3x4 arrangement. To describe this in Timeloop, an additional *Dummy-Buffer* is implemented and the *meshX* directive is added. The dummy buffer is implemented with the *regfile* class, but is only intended as a routing layer, this is indicated using constraints, see Lines 4 and 12 in Listing 5.3. The bypass constraints indicates none of the datatypes should be stored at that layer and the temporal constraint reiterates that by setting all factors to 1. Us-

ing the dummy buffers, 3 rows are created. Using the *meshX* attribute, the 12 PEs are divided such that there are 4 in the x direction, again creating 3 rows of PEs. To ensure proper routing of data, additional architecture constraints are applied to map the RS dataflow from Eyeriss to the 3x4 array.

Listing 5.2: Timeloop architecture description of the original Accelerator [12], only relevant attributes are included. **YAML**

```yaml
architecture:
  subtree:
    - name: system
      local:
        - name: DRAM
          class: DRAM
      subtree:
        - name: eyeriss
          local:
            - name: shared_glb
              class: smartbuffer_SRAM
            - name: DummyBuffer[0..3]
              class: regfile
              attributes:
                meshX: 4
          subtree:
          - name: PE[0..11]
            local:
              - name: ifmap_spad
                class: smartbuffer_RF
                attributes:
                  memory_depth: 24
                  meshX: 4
                  read_bandwidth: 3
                  write_bandwidth: 3
              - name: weights_spad
                class: smartbuffer_RF
                attributes:
                  memory_depth: 288
                  meshX: 4
                  read_bandwidth: 3
                  write_bandwidth: 3
              - name: psum_spad
                class: smartbuffer_RF
                attributes:
                  memory_depth: 32
                  update_fifo_depth: 3
                  meshX: 4
                  read_bandwidth: 3
                  write_bandwidth: 3
              - name: mac
                class: intmac
                attributes:
                  meshX : 4
```

Going through the constraints, it can be found that the GLB allows parallel processing over the output height (OH) dimension, as all other dimensions are spatially constrained. The dummy buffer allows parallel processing over the kernel height (KH) and IC dimensions. Inside the PE,

with all spads, the temporal constraints and thus the access pattern of data from the higher hierarchy layers. Using temporal constraints, 1 ifmap is processed at a time. By relaxing the temporal constraints for the weights and psum spads, the respective IC and kernel width (KW), and OC loops can be processed for that PE. The KH dimension is the outermost loop for all spads and all temporal factors are set to 1, as the KH will be accumulated outside the PE. The mapping space is constrained using two sets of mapspace constraints; the access patterns of the RAMs are specified using arbitrary but sensible orderings.

Listing 5.3: Timeloop architecture constraints of the original Accelerator [12]. Except for the dummy buffer, bypass constraints are omitted. **YAML**

```yaml
architecture_constraints:
  targets:
  # Bypass
  - target: DummyBuffer
    type: bypass
    bypass: [Inputs, Outputs, Weights]
  # Higher level constraints
  - target: shared_glb    #Global Memory (X-axis)
    type: spatial
    permutation: OH OC IC OW B KW KH
    factors: OC=1 IC=1 OW=1 B=1 KW=1 KH=1
  - target: DummyBuffer   #DummyBuffer (Y-axis)
    type: temporal
    factors: KH=1 IC=1 OH=1 OC=1 B=1 OW=1 KW=1
  - target: DummyBuffer
    type: spatial
    permutation: KH IC OC OH OW KW B
    factors: OC=1 OH=1 OW=1 KW=1 B=1
  # Constraints inside PE
  - target: ifmap_spad
    type: temporal
    permutation: OC IC KW OH OW B KH
    factors: OC=1 IC=1 KW=1 OW=1 B=1 OH=1 KH=1
  - target: weights_spad
    type: temporal
    permutation: OC IC KW OH OW B KH
    factors: OC=1 B=1 OH=1 OW=1 KH=1
  - target: psum_spad
    type: temporal
    permutation: OC IC KW OH OW B KH
    factors: IC=1 KW=1 OH=1 OW=1 B=1 KH=1
# Mapping space constraints
mapspace_constraints:
  targets:
  - target: DRAM
    type: temporal
    permutation: CME NSRF
    factors: N=1 S=1 R=1 F=1 C=1
  - target: shared_glb
    type: temporal
    permutation: FCM NSRE
    factors: S=1 R=1 E=1 N=1
```

A different schedule is created for every layer of YOLOv4 Tiny. To analyse the result of the original Timeloop description, layer 0, as given in Listing 5.4, will be discussed. The ifmap is

loaded 4 rows at a time from DRAM, as this matches the number of columns in the PE-array. In the *DummyBuffer* layer, kernel rows are applied, one for each row of the PE-array. The psums can then be accumulated vertically, to form 1 ofmap.

Inside the PE, one plane with dimensions $KW \cdot IC$ (so 9 ifmap values for this layer) is accumulated for 32 different output channels. Using this *store* topology, an ifmap value is reused $KW \cdot OC$ times inside the PE. Using the GLB, it can be reused KH more times as it is loaded into the next row of the PE-array, but this does require the NoC. This means only partial local reuse of ifmaps is realised for this first layer. For this layer there are a total of 864 kernel values, which are divided over the rows of the PE-array. Each PE can store $864/KH = 288$ kernel values. So, a kernel value is stored once and then fully reused as all ifmap columns and rows are sequentially processed.

As this schedule is not flexible, the utilisation and reuse is not optimal for all layers. An example is layer 5, where the *Spatial-Y* dimension of the PE-array is used to process ICs (as $K = 1$, so it cannot be spread over that dimension). As layer 5 has 64 input channels—and all loop iterations should be homogeneous—only two rows of the array are used, instead of all three as the layer dimensions are not a multiple of 3. This results in only 66.67% hardware utilisation for the entire layer [12]. This inefficiency is introduced for every layer where $K = 1$, such that a different loop must be used for the *Spatial-Y* dimension which can not efficiently be divided by the three rows.

Listing 5.4: Timeloop schedule for layer 0 of YOLOv4 Tiny, based on the original constraints.

```
DRAM [ Weights:864 Inputs:521667 Outputs:1384448 ]
-------------------------------------------------
| for OH in [0:52)
shared_glb [ Inputs:11259 Outputs:26624 ]
-----------------------------------------
|   for OW in [0:208)
|     for OH in [0:4) (Spatial-X)
DummyBuffer [ ]
---------------
|       for KH in [0:1)
|         for KH in [0:3) (Spatial-Y)
ifmap_spad [ Inputs:9 ]
-----------------------
|           for KH in [0:1)
weights_spad [ Weights:288 ]
---------------------------
|             for KW in [0:3)
|               for IC in [0:3)
psum_spad [ Outputs:32 ]
------------------------
|                 for OC in [0:32)
```

Improved schedule

The original schedule provided good reuse of datatypes, but made inefficient use of the hardware for layers where $K = 1$. Reuse should remain high, whilst the loop mapped for spatial reuse should always be an integer multiple. The smallest ifmap dimensions are 13x13 and all larger dimensions are an integer multiple of that. Thus, the OH or output width (OW) loop can be used if the PE-array contains 13 elements. The OC dimension can also be used. This dimension is at least an integer multiple of 32, so an even number of PEs can be used in the

array. To remain in the same order of magnitude, 16 would be a valid number of PEs. If much more hardware becomes available in a new design, it would be possible to process 2 or more ifmap rows (OH) of 13 elements (OW) simultaneously, or 32 output channels (OC).

The Timeloop architecture description for the new design is similar to Listing 5.2, except for the removal of the dummy buffer and the *meshX* attribute (the new Timeloop description is provided in Listing A.2). The PE-array is made 16 elements wide, with the target of processing OCs in parallel. The original schedule implements the ifmap and kernel spads using BRAMs, which are a fixed size in the FPGA fabric. As these BRAMs are not dynamically allocated, it allows the accelerator to utilise the entire specified BRAM. Given the Zynq BRAM structure, this is half a 18kB BRAM for each datatype [30]. This allows the maximum size of the ifmap and kernel spads to be 18kB, significantly larger than the original, without increasing the hardware utilisation.

The ifmap spad processes 1 ifmap at a time, where the psum spad stores multiple OCs. The weights spad allows storage for the KW, KH and IC dimensions. These are all required dimensions to fully compute 1 ofmap. As psums are now accumulated in a PE, the dataflow is no longer row stationary. When moving to the next ofmap position for computations, kernel size (K) ifmap elements are dropped, of which only one has completely been used. The remaining $K-1$ values will have to be re-fetched later. The authors of Eyeriss describe this new dataflow as output stationary [7], as the psums remain stationary in the PE. The impact of changing the dataflow type is discussed in Section 5.3.1.

Listing 5.5: Timeloop architecture constraints of the improved Accelerator. Bypass constraints are omitted. **YAML**

```yaml
architecture_constraints:
  targets:
  # Global buffer
  - target: shared_glb
    type: spatial
    permutation: B KH KW IC OC OW OH
    factors: B=1 KW=1 IC=1 OW=1 OH=1 # Only branch out OC and/or KH
  #Ifmap_spad - Row Stationary
  - target: ifmap_spad
    type: temporal
    permutation: B IC OW OH KH KW OC
    factors: B=1 IC=1 OW=1 OH=1 KH=1 KW=1 OC=1
  #Weight_spad - Row Stationary
  - target: weights_spad
    type: temporal
    permutation: B OC OW OH KW KH IC
    factors: B=1 OC=1 OW=1 OH=1
  #Psum_spad - one ofmap position but of different output channels
  - target: psum_spad
    type: temporal
    permutation: B IC OW OH KH KW OC
    factors: B=1 IC=1 OW=1 OH=1 KH=1 KW=1
```

Using these constraints, the schedule in Listing 5.6 is created. Again layer 0 serves as an example. It adheres to the constraints and processes OCs in parallel using the array and accumulates an entire ofmap in the PE. For 1 ofmap, N_{of} ifmap values are required. For layer 0, $N_{of} = 27$ values, as is found for the number of values stored in the ifmap spad. As layer has 32 output channels, 2 OCs are computed in each PE. This requires 2 sets of kernels, which coincides with the number of weights in the spad. ifmap values are reused in the PE-array by

multicasting to 16 parallel PEs. Inside the PE, they are reused to compute 2 ofmaps. New ifmap values are streamed in to compute for a new ofmap position. As the concept of 1D convolutions from the RS-dataflow is no longer applied, reuse along either the KH or KW dimension can no longer be realised and ifmap values are buffered in the GLB. As layer 0 has a limited number of kernels, all of them can be stored in the PEs ($864/16 = 54$) and reused optimally. This allows all kernels to be reused for the full RF_k and the ifmaps for a factor $rf_{if} = KW \cdot OC$.

Using these constraints, schedules for all layers are create that utilise the hardware for 99.05% for a complete inference. The remaining inefficiency is not caused by sub-optimal scheduling, but by the (limited) bandwidth of the NoC and memories, which means that PEs are waiting for data during the first cycles of executing a layer, as data is loaded in. The GLB is exclusively used to store ifmaps and no longer for other datatypes. As psums are now accumulated entirely to form ofmaps, they do not have to be stored intermittently. As reuse for ifmaps is optimised, the bandwidth for the kernel NoC should be optimised to provide sufficient data to sustain optimal parallel processing, even during those early cycles.

Listing 5.6: Timeloop schedule for layer 0 of YOLOv4 Tiny, based on the new constraints.

```
DRAM [ Weights:864 (864) Inputs:521667 (521667) Outputs:1384448 (1384448) ]
-------------------------------------------------------------------------
| for OH in [0:208)
shared_glb [ Inputs:3753 (3753) ]
---------------------------------
|   for OW in [0:208)
|     for OC in [0:16) (Spatial-X)
ifmap_spad [ Inputs:27 (27) ]
-----------------------------
|       for KH in [0:1)
weights_spad [ Weights:54 (54) ]
--------------------------------
|         for IC in [0:3)
|           for KH in [0:3)
|             for KW in [0:3)
psum_spad [ Outputs:2 (2) ]
---------------------------
|               for OC in [0:2)
```

5.3 Network-on-Chip

Heinsius makes several recommendations to improve the performance of the CNN accelerator. One of the first observations is that the throughput of PEs is limited by the bandwidth of the networks providing the data, as becomes evident by a non-optimal utilisation of the PE [12, section 7.5]. It is unclear if the bottleneck is the channel to the external DRAM, or the fetching of data from the local GLB, but given the available bandwidth for the AXI-bus to the external DRAM[2] it is expected the bottleneck is the NoC. Both could be improved, Heinsius proposes multiple solutions to speed up the bus to the external DRAM and specifies that the implementation of the NoC transfers one parameter at a time, whereas [7] transfers 4 parameters simultaneously.

5.3.1 Theoretical bound network-on-chip bandwidth

To determine the allowed time to load all data without limiting the speed of the computations, a theoretical analysis is performed. With the improvement of the PE throughput, it now takes 3

[2]The bus-width is 32 bits and 8 busses are used in the current implementation. Each operates at the design's clock speed of 125MHz, providing an effective bandwidth of 4GB/s.

cycles to perform a MACC. The number of MACCs is known to be N_{lay}, which are executed on *P* parallel PEs. Using this information the number of cycles to compute 1 layer of a CNN can be calculated as:

$$\text{\#cycles} = \frac{N_{lay}}{P} = \frac{OH \cdot OW \cdot KH \cdot KW \cdot IC \cdot OC}{P} \tag{5.1}$$

To achieve optimal performance, data should always be available to perform these computations, otherwise the NoC bottlenecks performance and the PEs cannot be utilised optimally. In the worst case, the NoC is used to provide each operand for the MACCs individually. With a separate network for each datatype, this would require the GLB to provide one of each datatype every 3 cycles and the NoC to relay it in a pipeline with a throughput period of 3 cycles in parallel for the P PEs. To relax the constraints on the GLB and NoC, data is reused locally in the spads, such that the number of fetches for the same operand can be reduced.

At the least, all individual ifmaps and kernels must be loaded. If values are optimally reused, they need only be fetched once and can be reused later. It is known that an ifmap can be reused $\text{RF}_{if} = KH \cdot KW \cdot OC$ and kernels $\text{RF}_k = IH \cdot IW / S^2$ times (see Section 2.3). As indicated in Section 5.2.1, optimal reuse is not always realised. Thus, values must be fetched more than once, demanding a faster NoC. As such, the speed at which data must be fetched can be related to these optimal reuse factors, RF_{if} and RF_k, and the realised reuse rf_{if} and rf_k, which is derived from the improved schedule. Using this information, the required throughput (λ) of the NoCs for respectively ifmaps and kernels can be derived:

$$\lambda_{if} = \frac{RF_{if}}{rf_{if}} \cdot \frac{\text{\#cycles}}{\text{\#ifmaps}} = \frac{KH \cdot KW \cdot OC}{KW \cdot OC} \cdot \frac{\frac{IH \cdot IW}{S^2} \cdot K^2 \cdot IC \cdot OC}{P} = \frac{K^3 \cdot OC}{S^2 \cdot P} \tag{5.2}$$

$$\lambda_k = \frac{RF_k}{rf_k} \cdot \frac{\text{\#cycles}}{\text{\#kernels}} = \frac{\frac{IH \cdot IW}{S^2}}{OH \cdot OW} \cdot \frac{\frac{OH \cdot OW \cdot K^2 \cdot IC \cdot OC}{P}}{K^2 \cdot IC \cdot OC} = \frac{OH \cdot OW}{P} \tag{5.3}$$

A sanity check verifies that for a higher number of parallel executing PEs, the NoC needs to provide more data and thus should provide higher throughput. These expressions also indicate that for higher reuse factors rf_{if} and rf_k, the throughput requirement is lowered. The reuse factors are fixed for a given schedule. For the improved Accelerator this is the schedule from Section 5.2.1. Given this realised reuse, which is constant for all layers, the least optimal layers can be found based on the layer parameters.

It was noted previously how layers where $K = 1$ are more stringent for data throughput. This is understated by Equation (5.2), where the resulting numerator only depends on OC. Layer 5 creates the highest demand for λ_{if} as it has the lowest number of output channels ($OC = 64$) in combination with this unit kernel size ($K = 1$). This results in a throughput demand of $\lambda_{if} = \frac{1^3 \cdot 64}{1^2 \cdot 16} = 4$. The throughput for kernels depends on the dimensions of the ofmap, which is smallest in layers 13–17, in which case the minimum required throughput is: $\lambda_k = \left\lceil \frac{13 \cdot 13}{16} \right\rceil = \lceil 10.5625 \rceil = 11$. When the NoC adheres to these bounds, it will not starve the PEs. For this analysis the start-up latency is not taken into account, which means that during the first cycles of executing a new layer, data is not directly available, resulting in temporary inefficient usage of the PEs. This is resolved during run-time when data is reused from the spads, allowing the NoC to catch-up fetching data.

5.3.2 Revision of implementation

Given the bounds derived in Section 5.3.1, the requirements of the NoC are known. The architecture of the PE-array has been adjusted to a 1 dimensional array, a presented in Figure 5.3.

The Catapult description is updated accordingly such that all design blocks from GLB to PE meet the throughput requirement. Data is loaded from the GLB using the respective AGENs and forwarded to a broadcast bus which provides the multicast controllers (MCs) in each PE-block with input values.

The broadcast bus is used since all hierarchical interfaces are implemented using *ac_channels* which can only have one producer and one consumer. Therefore, the broadcast bus reproduces the input for each outgoing channel, as described in Listing 5.7. The *for*-loop is completely unrolled such that the data is written to all output channels simultaneously. The `data` is of the `mc_data` datatype, which is a struct containing both the data, as well as an ID to represent the intended PE. The description is given on Lines 1 to 5 of Listing 5.7. The ID is compared in the multicast controllers inside the PE-block, the data is accepted if the ID matches and dropped from the *ac_channel* otherwise.

Listing 5.7: Catapult description of the broadcast bus. **C++**

```
1  template <typename dType, typename idType>
2  struct mc_data {
     dType data;
     idType id;
5  };
6
   template<class dType, class idType, int LEN>
   class Broadcast_bus {
   public:
10   Broadcast_bus(){}

   #pragma hls_design interface
     void run(ac_channel<mc_data<dType, idType> > &data_in,
              ac_channel<mc_data<dType, idType> > data_out[LEN]) {
15     if (data_in.size()>0){
         mc_data<dType, idType> data = data_in.read();
   #pragma hls_unroll yes
         for (int i=0; i<LEN; ++i) {
           data_out[i].write(data);
20       }
       }
     }
   };
```

5.4 GLB configuration

The NoC provides data to the PE in the array. It derives this data from the GLB which can provide storage for all datatypes and provides larger, albeit slower, local storage when compared to the spads. Moreover, by using the NoC, the GLB can provide data to all PEs, allowing for more reuse opportunities.

Inside the GLB, circular buffers can be created to provide storage for each datatype. Surrounding the GLB are address generators (derived from [18]), which are subdivided into two categories. First, there are the *fill* AGENs, which determine the order in which data is loaded from DRAM into the GLB. Secondly, there are *read* AGENs, which read data from the GLB and label it with an *ID* which is used to indicate what the intended PE(s) are. The IDs are compared by the MCs inside the PEs. This architecture is visualised in Figure 5.4.

A mapping is required that applies the schedule from Section 5.2.1 to these AGENs, as the

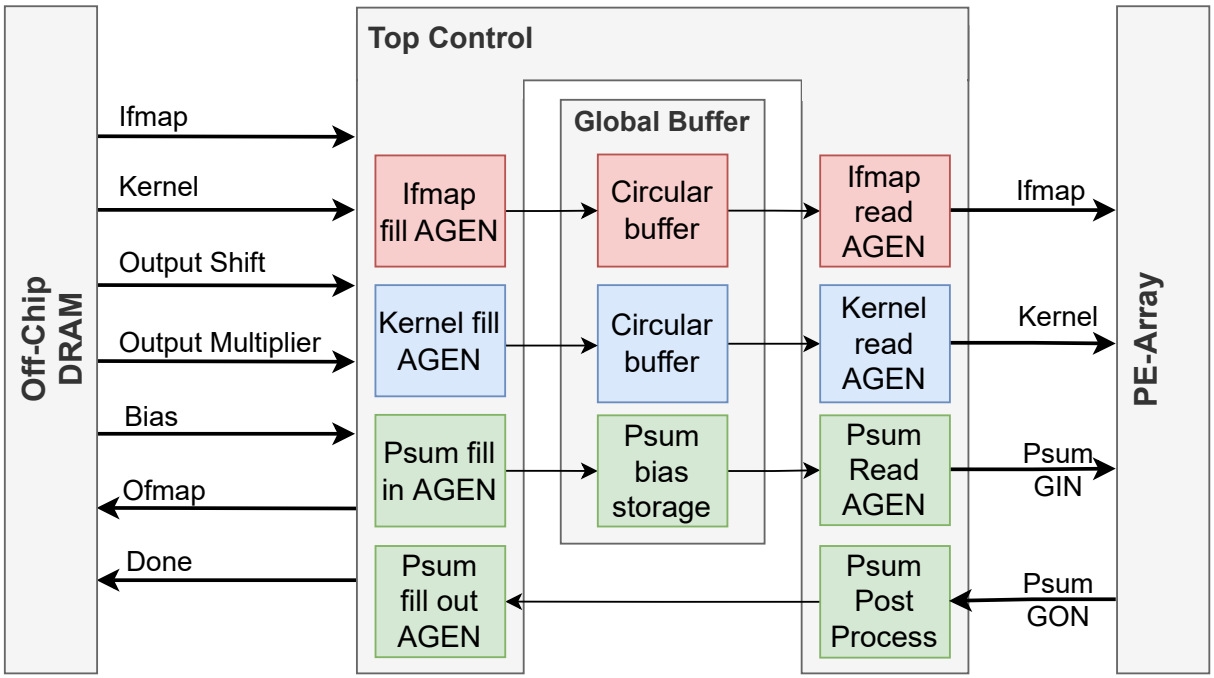


Figure 5.4: An overview of the GLB and surrounding AGENs.

order in which data is fetched also determines the order in which computations are performed. To achieve this, the text format schedule created using Timeloop (e.g. Listings 5.4 and 5.6) is parsed, such that the relevant parameters and loop order is obtained. As schedules and layer parameters vary between the layers of YOLOv4 Tiny, a separate configuration is created for each layer, the information must thus be generalised such that the final description can describe all layers. From this complete description, a C++ array of configurations is created that can be send to the Accelerator during run-time and forwarded to the AGENs, to indicate how data should be loaded.

This configuration controls the bounds of for-loops. To provide full flexibility for *all* schedules, a loop would be designated for every possible dimension and hierarchical level. As only a fraction of the possible loops is used by any given schedule, many loops would only iterate once as unused loops would be iterated at most 1 time. If this loop bound is static, it could be removed by the compiler. Given that a configuration is provided to the accelerator, it cannot be guaranteed that a loop will go unused during compile time. Hence, all loops are implemented in hardware, introducing unnecessary logic. This impacts both the hardware footprint and performance, which is why the implementation of loops in the Catapult description is not part of the automated configuration tool. The tasks that are implemented by the configuration tool is given in the next section.

5.4.1 Parsing schedule

A configuration tool exists for the original Accelerator, which is adjusted to be more generally applicable to any schedule. It parses Timeloop's text schedule and identifies the loop types with the dimensions from Table 2.1 and the hierarchical level according to Table 5.1. Five hierarchical levels are identified, level 1 represents the level between the first and second dimension of the PE-array and is exclusively used for the original Accelerator. It goes unused in the transition to the 1D array. The configuration tool has been adjusted, but maintains this layer to support 2D PE-array configurations in the future. To remain general, it is assumed that any dimension can have a loop at any hierarchical level.

Table 5.1: The hierarchical level in the schedule.

#	Description of level
4	Off-chip memory (DRAM)
3	Global buffer (SRAM)
2	Network-on-chip Spatial-X
1	Network-on-chip Spatial-Y
0	PE with all spads

As an example, the loops from the improved schedule for layer 0 of YOLOv4 Tiny in Listing 5.6 are extraced. The loops that are identified from that schedule are: *OH4, OW3, OC2, KH0, IC0, KH0, KW0, OC0*. It must be noted that the first KH loop only exists to provide any loop for the ifmap spad and is overwritten by the next occurrence of an KH loop at the PE level. If all layers are parsed, some additional loops are used: *OC4, OC3*. The first layer only has 32 output channels, whereas higher layers of the network have many more output channels, which are implemented at the GLB and DRAM level. Going forward it will be these loops that are implemented in the hardware, other combinations of dimension and level will be omitted.

5.4.2 AGEN loops in Catapult

To understand how the loops for the AGENs are defined, the usage of the different datatypes will be explored. This analysis is based on the improved schedule:

Ifmap These are broadcast to all PEs and loaded in the following order: KW→KH→IC→OW→OH. Such that for a given ifmap position, the values to compute the corresponding ofmap are loaded (KW→KH→IC).

Kernel As different OCs are computed at the different PEs, there is no opportunity for multicasting. Instead, kernel values are loaded round-robin for each PE. This occurs in the order: OC2→OC0→KW→KH→IC.

Psum These are computed in place, so only the bias values are loaded during the first initialisation. Again, these are loaded round-robin for each PE; so in order: OC2→OC0. This loop is repeated every time when the PEs start calculations for a new set of psums.

The *fill AGENs* are responsible for loading the data from the off-chip DRAM in the correct order in the GLB. Data is stored in the order corresponding to the Timeloop schedule. Given the loops in Section 5.4.2, it is possible to derive how the AGENs are programmed to load the different datatypes.

The *read AGENs* retrieve the data from the GLB and send it to the NoC. Data is loaded in order from the GLB, and labelled according to the Timeloop schedule to ensure that the data is received by the correct PEs.

5.5 Expected throughput improvement

Given the proposed solutions, an expected throughput improvement can be formulated. This is based on four components. First, the throughput of the individual PE will increase by a factor 6. Secondly, improved schedule makes better use of the available array. Heinius reports an overall utilisation of 66% [12], which reaches 99% with the improved schedule (see Section 5.2.1), a 1.5 times improvement. Thirdly, the array now contains 16 elements, to better fit the schedule. This increases the available computational power by 1.33 times. Finally, the NoC bottleneck is addressed which raises the efficiency of the array. Heinsius reported 22.68% efficiency [12],

which is largely influenced by the sub-optimal utilisation. As the new architecture adheres to the theoretical bounds from Section 5.3.1, the efficiency should rise to 100%, a 4.41 times improvement. Overall this yields a theoretical speed up of 52.9 times. Given this theoretical performance, inference is performed in 1.106 seconds.

6. RESULTS

6.1 Processing element throughput

The PE described in Section 5.1 is implemented in Catapult, the recorded latency and throughput values are recorded in Table 6.1. Additionally, to verify the performance gain, the improved PE could be simulated in Vivado. A test bench for this simulation can be created directly in Catapult. This way, all the relevant ports that Catapult infers are automatically connected to the test bench. Provide the PE as the design under test. Dummy data can be provided as test vector for the computations. It should follow a realistic schedule where $KH = 1$ to ensure results for old and new PE are equivalent.

The throughput of the original PE is 6 cycles, where the MACC operations requires 1 cycle for multiplication and 1 for addition, which are distributed between other PE operations. The improved PE realises a throughput of 1 MACC. This is thanks to the DSP primitive, and setting pipeline directives. Using the pipeline setting at most a throughput of once every two cycles could be achieved, by setting the *sharing allocation* to 0%, a schedule can be found that achieves a throughput of 1.

Achieving this also requires an improvement to the spads, which could not reach the desired throughput. Some of the improvements for the spad are given in Listing 6.1, with the full code available in Appendix A. The spad is implemented as an augmented circular buffer, with additional features to aid providing data during computations. A spad is instantiated to accommodate the largest size in the entire neural network; for smaller layers the `tempMaxSize` channel indicates the temporary upper bound of the memory. This allows looping around the circular buffer and loading data multiple times (e.g. the filter kernels). After computations for a row are finished, the spad can be reset, which is achieved by resetting the read and write pointers. The `data_in` and `data_out` ports are for inserting and retrieving data from the spad.

As found in Section 5.2.1, the datatypes are iterated in loops. Depending on how the loops are iterated and how data is stored in the spad, some values will have to be dropped freeing space for new elements. This is the case for ifmap values. Therfore, a different description is made for the ifmap spad which provides additional features. Using the `read_start_inc` channel, it becomes possible to indicate how many elements should be skipped to derive the memory location of the next ifmap position (thus taking into account the other dimensions, such

Table 6.1: The throughput results of the PE, derived using Catapult.

	Latency	Throughput
Test vector	3	1
MACC	2	1
PE controller	2	1
Ifmap spad	3	1
Other spad	3	1

as the IC number of elements that should be skipped). Moreover, given the improved schedule, the same ifmap value is used multiple times sequentially to be multiplied with filter values for different output channels. Hence, the read pointer for the spad should not shift over before all computations with it are performed. This is realised with a separate `read_inc` channel, which is driven by the PE controller to indicate when the pointer can be moved. For the other spads incrementing the pointer occurs automatically after each read.

On Line 14 of Listing 6.1 an if-statement is used to implement the wrap-around of the pointer, once it surpasses the (virtual) end of the buffer. This implementation can more effectively be converted to hardware, when compared to a modulo operation to achieve the same result. As such, the wrap-around is preferably implemented using this if-statement.

Listing 6.1: Improvements to the implementation of the spads inside the PE, complete implementation is found in Listing A.1. **C++**

```
SPAD(){ // Constructor for SPAD class
  static bool initValid = ac::init_array<AC_VAL_DC>(_spad,SIZE);
}

void run(//Inputs
      ac_channel<pointerType> &tempMaxSize,
      ac_channel<pointerType> &read_start_inc,
      ac_channel<bool> &reset,
      ac_channel<storageType> &data_in,
      //Output
      ac_channel<storageType> &data_out) {
  // Other logic omitted
  _read_start += read_start_inc.read(); //Update starting position SPAD
  if (_read_start > _max_size) { // If out of (virtual) bounds
    _read_start -= _max_size;
    _full = false; // As spaces are freed up, new data can be loaded
  }
}
```

The PE can achieve a throughput of 1 MACC per cycle, using the DSP48E1 primitive IP block. This provides an acceleration of 6 times for the processing of MACCs. These improved PEs can next be applied in the new PE-array.

6.2 Network-on-chip throughput

As most operations in the NoC are simple (e.g. reproducing channels in the broadcast bus, or comparing IDs in the MCs) the speed of the network is high. Pipeline directives are provided to the hierarchical blocks to meet the desired throughput. This is verified using the Catapult cycle report in Table 6.2. The table provides an overview of each hierarchical block and its latency and throughput measurement according to Catapult. The report shows difficulties in determining the throughput of the *psum_post_process* block. In this block, the psum originating from the PE-array is quantised before it sent back to the off-chip DRAM. The directives indicate that processing should occur in a pipeline with an interval of 4. As its throughput should not be the bottleneck for processing, the post-processing should meet the requirement that it can process ofmaps, as the PE-array produces them. This occurs every: $\frac{N_{of}}{P} = \frac{KH*KW*IC}{P}$ cycles. For YOLOv4 Tiny, the worst case is layer 5: $N_{of} = 1^2 \cdot 64$. Hence, the interval is 4 when $P = 16$.

In Table 6.2 the hierarchical blocks that are part of the path supplying the PE-array are highlighted. The table shows that each of these blocks provides sufficient throughput, as the minimum for ifmaps is 4 cycles, which is met, as each block has an initiation interval of 3 cycles.

Table 6.2: The performance result of the hierarchical blocks in the NoC, derived using Catapult.

	Latency	Throughput
Config	31	32
Broadcast bus	**1**	**3**
Multicast controller	**1**	**3**
PsumC out	1	3
PsumC in	**1**	**3**
MACC	2	1
Psum spad	3	1
Kernel spad	3	1
Ifmap spad	3	1
PE controller	2	1
Psum read AGEN	**1**	**3**
Ifmap read AGEN	**1**	**3**
Kernel read AGEN	**1**	**3**
Psum post process	269?	260?
Psum fill out AGEN	1	2
Ifmap fill AGEN	17	27
Kernel fill AGEN	16	26
Psum bias storage	4	7
Circular buffer	**3**	**4**

On the return path from the array, data can temporarily be stored in the ac_channels, such that it does not stall the computations. Hence, this result from Catapult demonstrates that the NoC bottleneck has been lifted.

6.3 Overall system performance

Verification with the individual PEs indicate that they perform the correct operations. They constitute the array inside of the Accelerator. The functional correctness of the overall array has *not* been verified. As its correct operation depends on the ordering of the loops, which would have to interchange order, the final implementation will not be largely different. The resource utilisation, too, should not be largely dissimilar from the current architecture.

The entire design is passed through the Catapult design flow and synthesised with Vivado, for a clock speed of 125MHz. Before synthesis the DSP IP-blocks are provided to the project such that the synthesis tool can include those in the designated black-box positions. After synthesis, Vivado provides an estimate of the resource utilisation, this is presented in Table 6.3. The utilisation is compared to the original implementation. The table demonstrates that the new design reduces LUT utilisation by 30.7% and the number of FFs by 20.5%. As the storage for psums is removed from the global buffer, the BRAM utilisation is also reduced. The remaining BRAMs are primarily used by the PEs. These reductions in utilisation are in the face of an increase in the array's size.

Noteworthy is the resource utilisation of the individual PE, which increases in the improved design, despite the use of the DSP. The improved design uses 228% more LUTs and 281% more FFs, as well as the additional DSP block.

To understand which parts of the design now require less resources, the utilisation is broken down into the individual hierarchical blocks in Table 6.4. In the original accelerator the top control amounted to 10798 LUTs and 11821 FFs [12, ch. 7]. The improved design uses a fraction of

Table 6.3: Resource utilisation of the original designs and the improved PE and complete Accelerator (post-synthesis). Utilisation by AXI components is not taken into account.

	LUT	FF	BRAM	DSP
Original PE	674	580	1.0	0
Original Accelerator	37934	47731	65.5	19
Improved PE	1539	1629	1.0	1
Improved Accelerator	26300	37924	19.5	29
Available on Zedboard	53200	106400	140	220

Table 6.4: Resource utilisation for the improved Accelerator (post-synthesis), broken down per hierarchical block.

	LUT	FF	BRAM	DSP
Config	1086	1426	0.0	0
Top control	3324	2868	3.5	13
PE-array	18697	23604	16.0	16
Other	3193	10026	0.0	0

those resources, a reduction of 324% and 412% respectively. The utilisation of the PE-array remains in the same order of magnitude, even though the size of the array has grown with 50%.

In Section 5.5 an estimation was made of the performance increase. From the presented solutions, the throughput of the PE is improved, the utilisation of the array is realised and the array is resized. As the functional correctness of the Accelerator is not validated, the efficiency cannot be measured. Given that NoC the bottleneck is successfully eliminated, the efficiency is assumed to be optimal. In this case, the Accelerator is able to perform 16 MACCs per cycle, at a clock speed of 125MHz. This yields a theoretical throughput of 2GMACC/s, or 4GOPS (as a MACC is 2 operations: multiplication and addition).

7. CONCLUSION

This work set out to answer the following research questions: *How can the throughput of a hardware accelerator for a CNN be improved to approach the theoretical upper bound using HLS for FPGA?* The sub-questions will first be answered to derive the final answer to the main research question:

- What is the theoretical upper bound for a CNN accelerator's performance on the Zedboard and how does this compare to performance in literature?
- Can the network-on-chip bandwidth bottleneck in the work by Heinsius be identified and resolved?
- Does a workflow with High-Level Synthesis influence the performance of a CNN accelerator?
- How can DSP-blocks be implemented in the Accelerator design using the high-level synthesis workflow?

In Chapter 3 different works were discussed. Specifically, alternative works implemented on the Zedboard were compared. It was found the Xilinx DPU implementation yields the highest indicative performance, with a peak throughput of 230GOPS. This serves as the upper bound for the performance of the accelerator.

In Section 5.3, a theoretical analysis was performed to determine the throughput requirements for the NoC. It was found that it depends on the layer characteristics as well as the number of PEs. In the worst case, the NoC providing ifmap values would need a pipeline initiation interval of 4, when 16 PEs are used. Several hierarchical blocks in the design by Heinsius did not meet this requirement. Hence, the NoC did prove a bottleneck for that design. In Table 6.2 the throughput periods for the different hierarchical blocks, as derived by Catapult, were presented. The blocks providing data to the PE-array are found to have a throughput period meeting the requirement. The blocks retrieving data from the array are not bound by a throughput requirement and the ac_channels connecting the different blocks provide a buffer to store ofmap values temporarily as the NoC sends them out.

Throughout the design Catapult HLS was used to create the architecture description. Using the custom datatypes, templates and pragmas, a proper description can be created for the accelerator. The tool lacks means to infer DSP-blocks directly in the design. So, instead an alternative work-flow was created to insert these after RTL was created in Catapult. By instantiating a black-box in the design, the option is left open to substitute it during synthesis. This is performed in Vivado, where the DSP48E1 primitive is inserted in the place of the black-box. Given this solution, a PE can be created that provides a throughput of 1 MACC per cycle, a 6 times improvement over the original PE by Heinsius.

Outside of DSP-block usage, no direct indication was found to suggest that Catapult of HLS in general provides a sub-optimal design. All designs implemented on the Zedboard in Section 3.3 were created using (Vivado) HLS. Using the tool's handles, control can be exerted over

timing/scheduling of designs. When correct coding practises are applied (as suggested by the user reference [23]), efficient hardware can be created. Using the possibility for high-level simulation accelerates debugging and optimising of the architecture, prior to synthesis. This could cut down on development times.

Using this information the main research question can be answered. Throughout this work multiple improvements for the Accelerator were proposed. Firstly, the previously discussed improvement for the PEs, which speeds up MACCs six times. Moreover, the structure of the Accelerator was adjusted to be more efficient and be better utilised based on the schedule created by Timeloop. The schedule divides the computations spatially over the PE-array. The array's size was increased to 16 elements to better accommodate the dimensions of YOLOv4 Tiny's layers. The array was simplified to be one dimensional, to reduce the NoC's complexity (and thus aid in resolving the throughput bottleneck). This leaves the spatial planning to the synthesis tool. With this updated schedule, the hardware utilisation is improved from 66% to 99%, a 50% increase. The efficiency of the Accelerator could not be derived without performing inference on the complete system on the Zedboard's SoC. The intended improvement of 4.41 times acceleration for the efficiency can thus not be validated.

The individual PE was functionally validated, whereas the entire Accelerator was not. Nevertheless, an indication can be provided of the Accelerator's resource utilisation. In Section 6.3, the synthesised Accelerator is compared to the work by Heinsius. The utilisation by the PEs has grown, despite the application of DSP-blocks to accelerate the MACC operation. The number of LUTs grows by 228% and the number of FFs increases almost threefold with 281%. Nevertheless, the entire Accelerator uses less resources, primarily thanks to the reduction in size of the top control. The storage for psum values was eliminated thanks to the improved schedule. Moreover, some of the AGENs were simplified, for instance the ifmap read AGEN, which can broadcast each ifmap value to all PEs. In total this results in a reduction of 30.7%, 20.5% and 70.2% of LUTs, FFs and BRAM respectively. As each PE now contains a DSP, but some of the hardware is more efficient, the number used grows from 19 to 29.

Concluding, this works presents a number of aspects that can, and have been, improved. These efforts have resulted in an architecture, which has not been fully functionally validated, but shows a promising outlook both in terms of throughput as well as resource utilisation. An average throughput of 4GOPS is estimated. This is 57.5 times slower than the Xilinx DPU implementation, but also 52.9 times faster than the implementation by Heinsius, which achieved a peak throughput of 0.12GOPS. Therefore, a considerable step is made towards the theoretically achievable throughput on the Zedboard platform.

8. DISCUSSION

The conclusion presents a promising architecture that provides gains in both throughput as well as resource utilisation. The latter indicates that more resources are available to provide further parallel processing, even on the Zedboard. The PE-array can be extended to contain more elements. However, the schedule should be taken into account to guarantee optimal utilisation of the hardware. As indicated in Section 5.2.1, the array is optimised to process OCs in parallel. The smallest layers two have 32 OC. Hence, the array can be scaled to 32 elements effortlessly. Increasing the number of PEs above this value will result in inefficiencies when utilising the hardware, if no alternative schedule is created. Additionally, the parallel processing can be extended by reinstating the RS-dataflow, where PEs forward psums such that each PE processes only 1 row of ifmap values. This improves the realised reuse factor for ifmaps (rf_if) to 100% reuse. This dataflow can also be achieved in the 1D array and could be inspired by the Chain-NN work. The local network from the work by Heinsius [12] is still available and can be applied for this purpose. This would require an adjustment of the configuration data to guide the flow and timing of psums. As well as an extension of the NoC, as the throughput demand grows further as the number of PEs grows (according to Equations (5.2) and (5.3)).

To characterise the current design however, some steps remain to be taken. Firstly, a full functional verification should be performed such that the Accelerator provides a bit-accurate replacement for the work by Heinsius. Afterwards, the accelerator should be deployed on actual hardware, in conjunction with the Tensorflow program. It could be deployed on the Zedboard to compare performance results and fully validate the increased throughput.

8.1 Recommendations

Based on the findings in this work, more opportunities for further acceleration can be explored. The expansion of the PE-array, as described previously, can be considered. With the comprehension of the scheduling, the array can be extended to process in parallel over multiple dimensions, primarily OC and KH.

In the design, a complete BRAM is used per PE, which has the same hardware cost as using only a fraction of the BRAM's storage capacity. However, only a fraction, 14.9%, of the available BRAMs are used. It might be feasible to assign multiple BRAMs to a single PE to provide larger local spads, without sacrificing throughput. The BRAMs available on the Zedboard and other Xilinx 7-series devices provide more opportunities for utilising the BRAMs, such as the current 2x18kB, 1x36kB or cascaded 1x64kB [30]. Larger spads allow more flexible scheduling and could allow for more (energy) efficient schedules for layers where there are many OCs and they are also scheduled at higher hierarchical levels of the architecture.

The Winograd algorithm and an application for NNs was discussed in Section 3.3. According to the work, the architecture achieved a peak throughput exceeding that of the Xilinx DPU, with 281GOPS. This indicates the opportunities with the Winograd algorithm. The authors of the work specify that the NN loses some accuracy, but there is no break down between bit-width

representation and the implementation of the Winograd algorithm. If the accuracy losses due to applying Winograd are conceivable, it might be a good candidate for further accelerating the MACC operations. This could be achieved by exploiting more of the DSP-block's capabilities, including the pre-adder to compute the psum terms from Equation (3.3). A condition is that the $k_1 + k_2$ term is pre-computed offline, to avoid increasing the number of filter kernels.

Besides the optimisations based on processing layers sequentially, the opportunity of fusing layers might also be explored. Exploiting inter-layer reuse may provide additional performance gains. It is known that the ofmap values from one layer serve as the ifmap values for the next. By keeping these on-chip, this saves bandwidth to and from the off-chip DRAM and possibly the NoC as data could even be kept locally in the spads.

This concept is further elaborated in [32], where the order of computations is changed in order to fuse layers. It is known beforehand which ifmap values from layer 0 correspond to the ofmap values from deeper layers. A schedule can be created which takes this inter-layer storage/reuse into account. OCCAM [10] was developed to derive such a schedule. The premise for this work is to find the conditions for optimal reuse of data, comparable to Section 2.3. Next the goal is to find and schedule a *dependence closure*, a sufficient condition for full reuse. The authors propose a dynamic scheduling algorithm to provide optimal inter-layer reuse which could be applicable to YOLOv4 Tiny too.

BIBLIOGRAPHY

[1] Manoj Alwani et al. “Fused-layer CNN accelerators”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783725.

[2] Arash Ardakani et al. “An Architecture to Accelerate Convolution in Deep Neural Networks”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.4 (2018), pp. 1349–1362. DOI: 10.1109/TCSI.2017.2757036.

[3] Chun Bao et al. “A Power-Efficient Optimizing Framework FPGA Accelerator Based on Winograd for YOLO”. In: *IEEE Access* 8 (2020), pp. 94307–94317. DOI: 10.1109/ACCESS.2020.2995330.

[4] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. “YOLOv4: Optimal Speed and Accuracy of Object Detection”. In: *CoRR* abs/2004.10934 (2020). arXiv: 2004.10934. URL: https://arxiv.org/abs/2004.10934.

[5] Kuo-Wei Chang and Tian-Sheuan Chang. “VWA: Hardware Efficient Vectorwise Accelerator for Convolutional Neural Network”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.1 (2020), pp. 145–154. DOI: 10.1109/TCSI.2019.2942529.

[6] Yu-Hsin Chen et al. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308. DOI: 10.1109/JETCAS.2019.2910232.

[7] Yu-Hsin Chen et al. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357.

[8] Robert David et al. “TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems”. In: *CoRR* abs/2010.08678 (2020). arXiv: 2010.08678. URL: https://arxiv.org/abs/2010.08678.

[9] Antonello Di Fresco and Giovanni Guasti. *Long Form Answer Record 73058: ResNet-50 CNN application implemented on a ZedBoard using Vivado and PetaLinux 2019.2*. Tech. rep. Xilinx, Nov. 2019.

[10] Ashish Gondimalla et al. *OCCAM: Optimal Data Reuse for Convolutional Neural Networks*. 2021. arXiv: 2106.14138.

[11] David Gschwend. “ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network”. In: *CoRR* abs/2005.06892 (2020). eprint: 2005.06892.

[12] L.R. Heinsius. *Real-Time YOLOv4 FPGA Design with Catapult High-Level Synthesis*. June 2021. URL: http://essay.utwente.nl/86465/.

[13] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017). URL: http://arxiv.org/abs/1704.04861.

[14] Hyoukjun Kwon et al. “MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings”. In: *IEEE Micro* 40.3 (2020), pp. 20–29. DOI: 10.1109/MM.2020.2985963.

[15] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1.

[16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[17] Angshuman Parashar et al. “Timeloop: A Systematic Approach to DNN Accelerator Evaluation”. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 304–315. DOI: 10.1109/ISPASS.2019.00042.

[18] Michael Pellauer et al. “Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration”. In: ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 137–151. ISBN: 9781450362405. DOI: 10.1145/3297858.3304025. URL: https://doi-org.ezproxy2.utwente.nl/10.1145/3297858.3304025.

[19] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[20] Colin Shea, Adam Page, and Tinoosh Mohsenin. “SCALENet: A SCalable Low Power AccELerator for Real-Time Embedded Deep Neural Networks”. In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. GLSVLSI ’18. Chicago, IL, USA: Association for Computing Machinery, 2018, pp. 129–134. ISBN: 9781450357241. DOI: 10.1145/3194554.3194601. URL: https://doi.org/10.1145/3194554.3194601.

[21] Siemens. *Catapult High-Level Synthesis and Verification*. Tech. rep. v2021.1_1. Siemens Digital Industries Sofware, Nov. 2021.

[22] Siemens. *Catapult® Synthesis HLS Bluebook*. Tech. rep. v2021.1_1. Siemens EDA, Nov. 2021.

[23] Siemens. *Catapult® Synthesis User and Reference Manual*. Tech. rep. v2021.1_1. Siemens EDA, Nov. 2021.

[24] Vivienne Sze. “How to Evaluate Deep Neural Network Accelerators”. Conference on Computer Vision and Pattern Recognition. 2020.

[25] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. “Scaled-YOLOv4: Scaling Cross Stage Partial Network”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2021, pp. 13029–13038.

[26] Shihao Wang et al. “Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks”. In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. 2017, pp. 1032–1037. DOI: 10.23919/DATE.2017.7927142.

[27] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. “Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs”. In: *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2019.

[28] Xilinx. *7 Series DSP48E1 Slice, User Guide*. Tech. rep. UG479 (v1.10). Xilinx Inc., Mar. 2018.

[29] Xilinx. *DPUCZDX8G for Zynq UltraScale+ MPSoCs*. Tech. rep. v3.3. Xilinx Inc., July 2021.

[30] Xilinx. *DSP48 Macro v3.0, LogiCORE IP Product Guide*. Tech. rep. PG148. Xilinx Inc., Nov. 2015.

[31] Zhewen Yu and Christos-Savvas Bouganis. “A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Fernando Rincón et al. Cham: Springer International Publishing, 2020, pp. 330–344. ISBN: 978-3-030-44534-8.

[32] Chen Zhang et al. “Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060.

A. CATAPULT IMPLEMENTATION

This work discusses a number of the hierarchical blocks in the Catapult design of the Accelerator, and provides several Timeloop descriptions. The complete code is not presented in the relevant sections, as it contains irrelevant sectoins for the provided examples. For completeness, the complete code for these topics is presented in this chapter.

A.1 Catapult

Listing A.1: Improvements to the implementation of the spads inside the PE. **C++**

```
#include <ac_channel.h>
#include <ac_int.h>
#ifndef CATAPULT_INDEPENDENT
#include <mc_scverify.h>
#endif

template<class storageType, int SIZE>
class SPAD
{
private:
  //Circular buffer
  storageType _spad[SIZE];

  //Pointers
  typedef ac_int<ac::nbits<SIZE>::val,false> pointerType;
  pointerType _read_index;
  pointerType _write_index;
  pointerType _max_size;
  bool _full;

  void reset_ptr() {
    _read_index = 0;
    _write_index = 0;
    _full = false;
  }

  void increment_ptr(pointerType* itr) {
    if (*itr < _max_size) {
        (*itr)++;
    } else {
        *itr = 0;
    }
  }

public:
```

```
  SPAD(){
    static bool initValid = ac::init_array<AC_VAL_DC>(_spad,SIZE);
    if (!initValid) {
      std::cout << "Initialisation of SPAD failed!" << std::endl;
    }
    reset_ptr();
  }

  #pragma hls_design interface
#ifndef CATAPULT_INDEPENDENT
    void CCS_BLOCK(run)(
#else
    void run(
#endif
            //Inputs
            ac_channel<pointerType> &tempMaxSize,
            ac_channel<bool> &reset,
            ac_channel<storageType> &data_in,
            //Outputs
            ac_channel<storageType> &data_out) {

    // Reset pointers if system is reconfigured
    if (reset.size() > 0) { // See if a new value is available
      reset.read(); // A value in the channel indicates a reset, actual
          value not relevant
      reset_ptr();
    }

    // Find if there is a new max size
    if (tempMaxSize.size() > 0) {
      _max_size = tempMaxSize.read();
    }

    // Write if new data is available
    if (data_in.size() > 0){
      _spad[_write_index] = data_in.read();
      increment_ptr(&_write_index);
      _full = _write_index == _max_size;
    }

    // Read if indices are not equal (e.g. don't read unwritten positions)
    // Except when the SPAD is full, in which case a read can safely be
        performed of every element
    if (_full || _write_index != _read_index) {
      data_out.write(_spad[_read_index]);
      increment_ptr(&_read_index);
    }
  }
};
```

A.2 Timeloop

A new architecture description was made for the 1D PE-array. This omits the *meshX* attribute and completely removes the *DummyBuffer* hierarchical layer compared to Listing 5.2.

Listing A.2: Timeloop architecture description of the improved Accelerator. **YAML**

```yaml
architecture:
  subtree:
    - name: system
      local:
        - name: DRAM
          class: DRAM
          attributes:
            type: LPDDR3
            width: 64
            block-size: 4
            word-bits: 16
      subtree:
        - name: accelerator
          local:
            - name: shared_glb
              class: smartbuffer_SRAM
              attributes:
                memory_depth: 30000
                memory_width: 32
                n_banks: 112
                block-size: 1
                word-bits: 8
                read_bandwidth: 3
                write_bandwidth: 3
          subtree:
          - name: PE[0..15]
            local:
              - name: ifmap_spad
                class: smartbuffer_RF
                attributes:
                  memory_depth: 18432 # Half a RAM-block
                  memory_width: 8
                  block-size: 1
                  word-bits: 9
                  read_bandwidth: 1
                  write_bandwidth: 1
              - name: weights_spad
                class: smartbuffer_RF
                attributes:
                  memory_depth: 18432 # Half a RAM-block
                  memory_width: 8
                  block-size: 1
                  word-bits: 8
                  read_bandwidth: 1
                  write_bandwidth: 1
              - name: psum_spad
                class: smartbuffer_RF
                attributes:
                  memory_depth: 16
                  memory_width: 20
                  update_fifo_depth: 3
```

```
        block-size: 1
        word-bits: 20
        read_bandwidth: 1
        write_bandwidth: 1
    - name: mac
      class: intmac
      attributes:
        datawidth: 8
```