FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Sparsity Aware Atrous Convolution Accelerator

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Fabian Kresse
Matrikelnummer 11707724

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.-Ing Muhammad Shafique
Mitwirkung: Univ.Ass. Muhammad Abdullah Hanif

Wien, 13. September 2021 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Sparsity Aware Atrous Convolution Accelerator

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Fabian Kresse
Registration Number 11707724

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr.-Ing Muhammad Shafique
Assistance: Univ.Ass. Muhammad Abdullah Hanif

Vienna, 13. September 2021

(Signature of Author) (Signature of Advisor)

Acknowledgements

I want to thank my advisor Prof. Shafique for providing me with the opportunity to work on this very interesting and exciting topic. Furthermore, I want to thank Univ. Ass. Hanif for proof-reading and providing me with valuable feedback over the course of this thesis.

Abstract

Various accelerators for atrous convolutional neural networks have been developed (e.g. [1, 2, 3, 4]). Only the authors of [4] consider dynamic sparsity due to zero input values and weights. However, since many implementation details are missing in [4], an efficient atrous convolution accelerator that can handle dynamic sparsity is proposed and implemented in this thesis. The proposed accelerator is based on the ZeNA accelerator [5] for standard CNNs. The implementation details are published along with the source code of the accelerator in this work.

Contents

1 Introduction **1**

2 Background **5**
2.1 Artificial Neural Networks 5
2.2 Convolution Neural Networks 7
2.3 DeepLabv3 11
2.4 Hardware Acceleration Techniques 12
2.5 Atrous Neural Network Accelerators 17
2.6 FPGAs (ZCU104) 19

3 Problem Statement **21**

4 Accelerator Architecture **23**
4.1 Overview 23
4.2 Dataflow 24
4.3 Control Unit 28
4.4 Bitvec Generation Unit 32
4.5 Processing Element 33
4.6 Requantization Unit 34

5 Results and Discussion **37**
5.1 Result Generation 37
5.2 Effect of Pruning on Accuracy 40
5.3 Results & Discussion of the Runtime 41
5.4 FPGA Power and Resource Utilization Results 47
5.5 Comparison with [2] 49

6 Future Improvements **51**
6.1 Improving Utilization 51
6.2 Increasing number of PEs 53
6.3 Obtaining a full-fledged Accelerator 53
6.4 Removing Exploitation of Dynamic Sparsity 55

7 Conclusion **57**

Bibliography **59**

List of Abbreviations **65**

A README **67**
A.1 Running the accelerator 67

B Source Code **69**
B.1 VHDL 69
B.2 Python 146

CHAPTER 1

Introduction

Artificial Intelligence (AI) has deeply intrigued humans since well before modern times. The first known stories of AIs date back as far as 600 BCE when Greek poets like Homer and Herodot imagined figures distinctly artificial, but also definitely intelligent, like Pandora and Talos [6]. While most contemporary AI researchers believe that such human like intelligence is at least a few decades away [7], significant progress has been made in the field of AI since the research kickoff into modern artificial intelligence in the 1960s.

While initial contributions to the AI field were able to solve and propose solutions for many problems that are hard for humans (at least on large scales) e.g. logical reasoning based on hand-crafted rules, they performed poorly at others that can be handled easily by even a four year old human child, such as vision problems [8]. Initially it was believed that these intuitively easy problems could be solved within the first decades of research, this turned out not to be the case and research fields like *computer vision* have seen a huge amount of interest since then [9].

Like many other AI problems computer vision systems have many practical applications and have therefore been investigated intensively since the early 1970s. The first approaches mainly relied on the 'Good Old-Fashioned Artificial Intelligence' [1] approach [10]. This approach to computer vision was mostly led by utilizing hard-coded human knowledge and completely human designed and tuned algorithms. These algorithms encompass still popular techniques like edge detection, edge labelling and some statistical methods. For a more detailed discussion the reader is referred to [9]. However, these approaches have faced difficulties to provide sufficiently good solutions for many problems [8].

More recently *machine learning* algorithms started conquering AI research and have yielded massive breakthroughs in fields like computer vision. In machine learning patterns are extracted from raw data and the outcome of the *inference* no longer solely depends on human encoded knowledge but is rather *learned* by the algorithm [8]. The conquest of machine learning was mainly enabled by a substantial increase in computing power, an improvement in algorithms (e.g

[1] As coined by Haugeland in his book 'Artificial intelligence: the very idea'

the introduction of *back-propagation* for Neural Networks [11]) and the more readily availability of large amounts of (labelled) data [8].

One class of machine learning algorithm that have greatly benefited from the improvements in the last decades and have subsequently surged in popularity, are biologically inspired *artificial neural networks* (ANNs) (for a more detailed discussion see chapter 2.1). While first being envisioned in the 1940s, the usage of neural networks saw very limited use due to limited computational resources and not yet advanced enough algorithms. With the increases in parallelization and computation power by GPUs, many-core CPUs, and specialized hardware like TPUs, ANNs have yielded ground-breaking results in a variety of disciplines.

Especially *Convolutional neural networks* (CNNs) have seen widespread use in computer vision and have been the de facto standard in recent years. This recently has been challenged by transformers, which have beaten CNNs on a variety of benchmarks (e.g. [12, 13] and [14]). However, CNNs will probably remain of interest in at least the foreseeable future for a variety of reasons, ranging from requiring less training data to joined CNN-Transformer architectures that are being investigated (e.g [15]).

One of the computer vision problems that CNNs have been applied to is *semantic image segmentation*. In semantic image segmentation the neural network tries to group individual pixels that belong to the same object and subsequently labels such grouped pixels [9]. To improve the segmentation accuracy different convolution types have been used, one of the currently best performing models DeepLabv3, uses a ResNet backbone and subsequently applies atrous convolution, a special type of the convolution operation to increase long range feature detection [16]. Semantic image segmentation sees use in a large variety of established industries e.g. medical imaging, traffic management, as well as in new ones, like autonomous driving.

While many of these applications do not have tight latency and throughput requirements some important ones, like autonomous driving, require them. Furthermore, energy-efficiency is a concern in almost every application and even more so in embedded and edge devices. Since the IoT-Trend has really taken off, the number of embedded and edge devices has exploded drastically and many of them are deployed for computer vision tasks.

Problematically, while CNNs yield good results, they are very computation intensive. However, computing the output of a neural network can easily be parallelized. While traditional dense 32-bit floating-point CNNs can be mapped very well on GPUs, more recent advanced techniques (for an overview see e.g. [17, 18] or [19]) that introduce irregular parallelism and reduce the bit precision cannot be mapped efficiently [20]. Furthermore, specialized convolutions that further decrease efficiency on GPUs like *atrous* (or *dilated*) convolution have been introduced. Because of these reasons it is paramount to use specialized hardware for CNNs to keep both latency and energy requirements low. This especially holds for embedded devices where both aforementioned resources are scarce. Therefore, application-specific integrated circuit (ASICs) and field-programmable-gate-arrays (FPGAs) have been deployed and used as *neural network accelerators*. ASICs have a much higher entry barrier due to initial manufacturing costs, longer time-to-market and do not offer the benefit of being re-configurable to incorporate later advances. However, often the benefits of developing an ASIC trump the downsides and many ASIC neural network accelerators have been developed. Many of them focus on different CNNs and convolution types e.g. [5, 21]. However, some are more general purpose and provide

support for a wider variety of convolution types: e.g. [4].

Due to rapid algorithmic advances, FPGAs are often used as testbeds for ASICs or as the deployed end-product. FPGA vendors have recognized the need for AI on FPGA solutions, and various tools to speed up development of FPGA based AI have become available [22, 23]. While FPGA based neural network accelerators have been lagging behind GPUs in terms of performance for some time, algorithmic and technological improvements have made them more competitive recently in instances where high performance and not only energy efficiency is desired [18, 20].

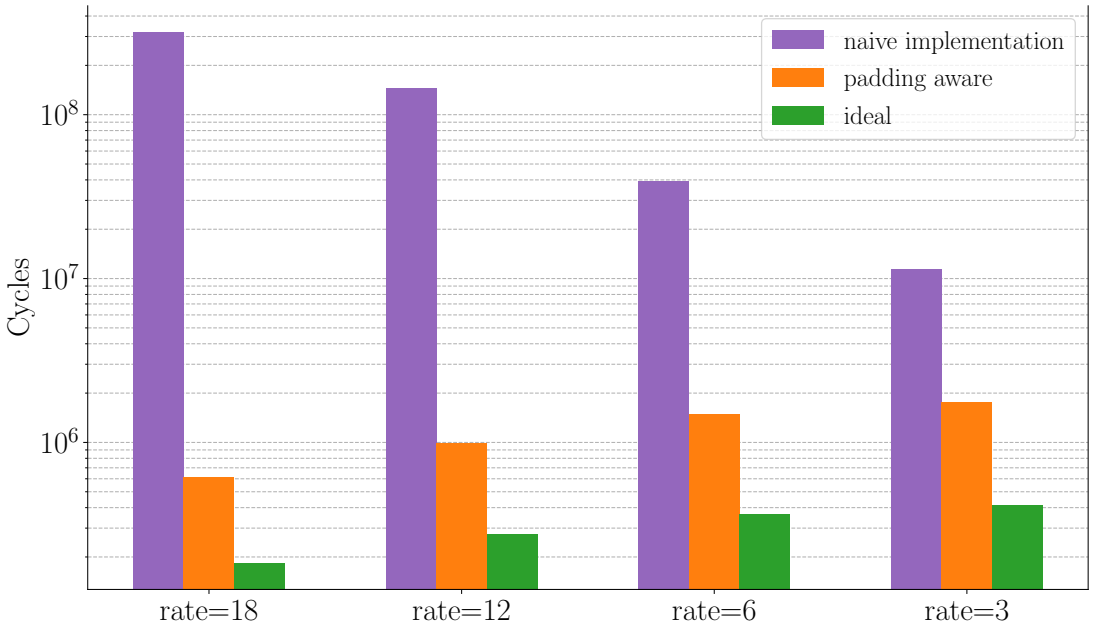

Figure 1.1: Number of multiplications needed to process one specific input ifmap with atrous convolution for varying atrous rates and implementation approaches. Note that the y-scale is logarithmic and only 23% of multiplications are valid. Even though only a specific ifmap with size 33×33 is shown, the differences translate also to other ifmaps.

This thesis proposes and implements a sparsity aware, 8-bit quantized FPGA-based accelerator for *atrous* convolution on a Xilinx ZCU104 FPGA.[2] While there already exists a large variety of accelerator for different CNNs (e.g. [21]), and many even support atrous convolution (e.g. [1, 2, 3, 4]), the implementation in [4] is the only one known to the author that implements sparsity aware atrous convolution that not only leverages the sparsity introduced by the structure of atrous convolution (for a detailed discussion see Chapter 2.2), but also the sparsity introduced by activations and weights that are zero. Additionally, [4] does not provide many implementation details or code. Because of this, the accelerator proposed here is based on ZeNA (zero aware neural network accelerator), a sparsity aware accelerator for conventional CNNs [5]. Mainly the dataflow of the accelerator was modified to allow efficient processing of atrous convolution. As

[2]The full source code is available on github: [24].

can be seen in Figure 1.1 a naive implementation of an accelerator for atrous convolution takes many more cycles to compute the result of one layer than would be needed in the ideal case. While implementations that only leverage sparsity due to the structure of the problem can at most obtain a speedup labelled in the figure as 'padding aware' the implementation of [4] and this thesis can come closer to the ideal case.

This thesis is organized as follows: Chapter 2 discusses the necessary background of neural networks and hardware-based acceleration. Especially CNNs and atrous convolution are highlighted. Chapter 3 gives the exact problem definition and lists the goals of this work. Chapter 4 describes the accelerator architecture in detail. Chapter 5 discussed the accelerator performance for various configurations. Chapter 6 discusses various improvements that could be added to the accelerator. Chapter 7 concludes the thesis and lists the contributions of this work.

CHAPTER 2

Background

This chapter gives an introduction to artificial neural networks (ANNs) and subsequently dives into convolution neural networks (CNNs). CNNs are especially popular in many AI applications, e.g. in computer vision. A special type of convolution, so called atrous or dilated convolution, is important in the context of this thesis and is explained later in this chapter. Additionally, this chapter highlights some typical techniques used in neural network accelerator designs. Especially techniques utilized in this thesis are discussed. At the end of this chapter the ZCU104 FPGA is shortly discussed.

2.1 Artificial Neural Networks

As the name already implies ANNs are heavily inspired by biological neurons such as in the human brain. In 1943 McCulloch and Pitts formulated a mathematical model of such a neuron [25]. A simplified neuron has a set of inputs I_i and an output O. The I_i's are multiplied by a weight vector yielding a vector-vector multiplication. The result is subsequently passed to an activation function g that returns the final output O:

$$g(\begin{bmatrix} I_1 & ... & I_n \end{bmatrix} \begin{bmatrix} w_1 \\ ... \\ w_n \end{bmatrix}) = O \tag{2.1}$$

There are various activation functions g that can be used for neural networks. One that is very popular and utilized in the context of this thesis is the *rectified linear unit* (ReLU) (e.g. [16]). The ReLU function satisfies the following Equation:

$$g(x) = \begin{cases} x & x > 0 \\ 0 & otherwise \end{cases} . \tag{2.2}$$

To form a *neural network* multiple *layers* of neurons are appended after each other such that the outputs of one layer form the input of the next layer. The most basic form of a layer is the

fully connected layer (FC layer) where every output of a neuron in the current layer is an input for every neuron in the next layer. Accordingly, Eq. 2.1 can be extended to one vector-matrix multiplication per layer, where each column in the matrix constitutes one neuron weight vector:

$$g(\begin{bmatrix} I_1 & ... & I_n \end{bmatrix} * \begin{bmatrix} w_{11} & ... & w_{m1} \\ ... & ... & ... \\ w_{1n} & ... & w_{mn} \end{bmatrix}) = \begin{bmatrix} o_1 & ... & o_m \end{bmatrix} \tag{2.3}$$

The first layer, also called the *input layer*, takes data input e.g. an image or an encoded audio signal and the last layer (the *output layer*) outputs the final *logits*. The form of this output depends on the architecture of the last layer. Layers that are in between the first and last layer are called *hidden layers*. The total amount of layers is revered to as the *depth* of the neural network. Figure 2.1 shows such a simple neural network consisting of only FC layers.

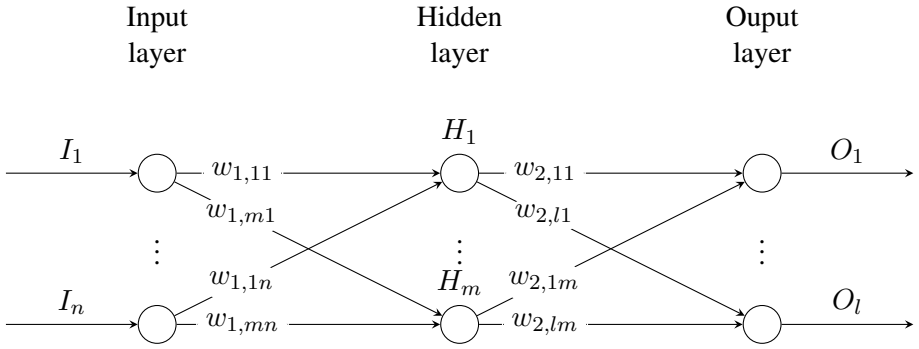

Figure 2.1: A simple neural network consisting of only FC layers. The first number labeling the weights indicates the layer, the second the position in the matrix of the layer. m, n and l denote the number of neurons in the respective layers. Figure adapted from [26].

The weight vector w_i that is used in Eq. 2.1 needs to be *learned* before deployment of the neural network. There are various algorithms to learn the per neuron weight vector, such as through genetic algorithms (e.g. [27]), Hebbian learning [28] and back-propagation [11]. The by far most popular approach is through back-propagation as introduced by Rumelhart, Hintont and Williams [11]. Elaborations about how back-propagation works can be found in all standard deep learning and AI textbooks (e.g. [8, 29]). The topic will not be further discussed in this thesis. This thesis mostly focuses on the *inference* part of the process, meaning that all weights have already been computed and are ready for deployment in an application.

As pointed out by Goodfellow et.al., the essential power of neural networks lies in the fact that they are efficient nonlinear function approximators [8]. Furthermore, since the function approximations are learned through data rather than being hard-coded, little to no human expert knowledge of the application field is required. This enables deployment of neural network in a vast variety of fields. However, most applications are best served by neural networks architectures that feature special layers. One such layer architecture that has seen tremendous success is the convolution neural network as discussed in the next section.

2.2 Convolution Neural Networks

Convolutional neural networks [30] are heavily inspired by neuroscience. Specifically, CNNs take heavy inspiration from the *primary visual cortex* (short V1), a region in the brain that is responsible for processing images. CNNs try to mimic this region, nevertheless there are significant architectural and hence also performance differences between the human brain and CNNs. In part this is due to the fact that many aspects of the human vision system are still not fully understood, a short overview of some key points is given in [8]. However, CNNs that are deployed for vision tasks can still yield very good results and can be comparable to, or even outperform, humans on limited vision tasks [8, 31]. Therefore, CNNs see broad practical application in computer vision and other tasks that require similar spatial information processing (like *natural language processing*).

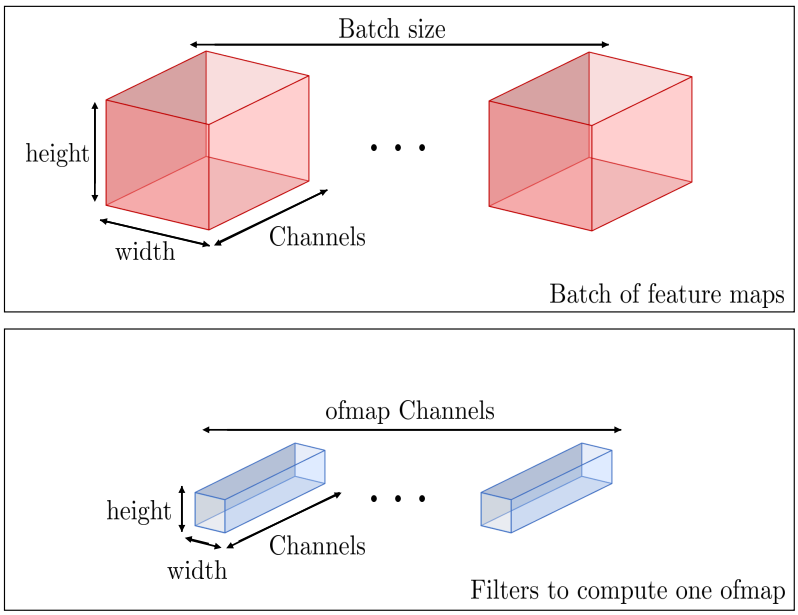

Figure 2.2: Dimensions of feature maps and filters.

The basic building block of a CNN is the *convolutional layer*. The input to a convolutional layer is called the *input feature map* (*ifmap*), analogously the output is called the *output feature map* (*ofmap*). In a 2D context (e.g. image processing) the *ifmap* and *ofmap* are typically tensors that are comprised of four dimensions: the batch size, channels, width, and the height. The different dimensions are shown in Figure 2.2.

The batch size determines how many input images are computed in parallel and can be an important parameter for neural network accelerators. For the accelerator developed in this thesis the batch size is always assumed to be one. The different channels encode *features* of the original input image. Depending on the input images dimensions, the channels in the first layer may for example correspond to the RGB components. In later layers more abstract properties, like edge patterns, are encoded in the channels.

In the initial layer the width and height correspond to the input images width and height. However, due to special layer architectures the width and height of the initial input image are reduced throughout the CNN. Commonly used operations in CNNs that exhibit such a down sampling property are *pooling layers* and the *stride* operation.

Whereas in a FC layer every neuron has one weight for each ifmap value, in CNNs so called *filters* (consisting of weights) are utilized to describe the connection between layers. Generally, filters are three-dimensional, and similarly to *ifmaps* and *ofmaps* have a width, height, and channel dimension. While the number of channels in the filter is the same as in the ifmap the width and height of the filter are generally much smaller [8]. Additionally, each filter is utilized to compute a single output channel, therefore the number of needed filters is equivalent to the number of ofmap channels (this is further discussed in the next section). In this thesis the 2D filter components that span only the width and height will be called *kernel*.[1]

Moreover, the values that make up an ifmap will be called *input activation values* (*iacts*). Figure 2.2 visualizes the different dimensions of filters and feature maps.

The convolution operation

CNNs perform an operation called *convolution* where the ifmap is convolved with the filters to compute the channels of the ofmap. To perform the convolution operation for a single channel the corresponding kernel's field of view (FOV) is moved over the ifmap. Every kernel has a value for each discrete spatial point in the FOV, the weight. Each weight is multiplied with its spatially corresponding iact. Subsequently, the results from the multiplications in the current FOV are summed up to yield the pre-activation function ofmap value. To compute the next value, the FOV of the kernel is shifted and the operation is repeated until all pre-activation function ofmap values have been computed. An example convolution for a 3×3 kernel (width and height equals three) is shown in Figure 2.3.

The convolution operation for one channel of a 2D ifmap with a 2D kernel of size $k \times k$ can therefore be written as:

$$ofmap(x,y) = g(\sum_{i=\lceil -\frac{k-1}{2} \rceil}^{\lceil \frac{k-1}{2} \rceil} \sum_{j=\lceil -\frac{k-1}{2} \rceil}^{\lceil \frac{k-1}{2} \rceil} ifmap(x+i, y+j) * kernel(i,j)) \tag{2.4}$$

Henceforth written as:

$$ofmap(x,y) = conv(ifmap(x,y), kernel) \tag{2.5}$$

To compute the ofmap for multiple ifmap channels, the pre-activation result for each channel is first computed as shown in Eq. 2.4. Next, the results of the different channels are added together and passed through the activation function to compute one channel of the ofmap. Therefore, to also compute a number of ofmap channels, the complete filters of one layer have dimensions (*ifmap channels*, *ofmap channels*, *width*, *height*). Hence, with o_c being the ofmap channel,

[1] Generally the term filter and kernel are often taken to be interchangeable, the terminology varies in this respect.

i_c the ifmap channel and $I_{c_{total}}$ the total i_c's, Eq. 2.4 can be extended for multiple channels as follows:

$$ofmap(o_c, x, y) = g(\sum_{i_c=0}^{I_{c_{total}}-1} conv(ifmap(x, y, i_c), kernel(o_c, i_c))) \tag{2.6}$$

As can be seen from Eq. 2.4 special handling at the edges of an ifmap is required. This is done by padding the ifmap with zeros around the edges. The size of this padding controls the width and height of the ofmap. During this thesis the most relevant padding is the so called *'same'* padding, where the ofmap retains the same width and height dimension as the ifmap.

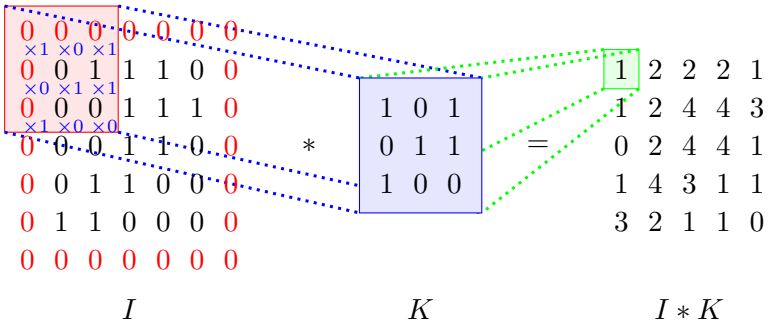

Figure 2.3: An example convolution operation. The filter K is moved over the ifmap I to yield the pre-activation function ofmap $I * K$. The 'same' padding of the ifmap is shown as red 0's. Figure adapted from [32].

Properties of CNNs

As elaborated upon by Goodfellow et.al. there are some key reasons why convolutional layers often outperform FC layers in many aspects such as number of calculations vs. accuracy [8]: First, the number of weights and operations is greatly reduced in comparison to FC layers since the neurons are much more sparsely connected. Only inputs in the same spatial vicinity are connected to neurons in the next layer. The size of this vicinity is determined by the width and height of the filter. However, spatial far away neurons can still interact *indirectly* over the subsequent layers and therefore CNNs can still yield great generalisation performance. Second, not only connections between the layers are reduced, also the number of different weights is greatly reduced (i.e. many of the connections share the same weights). While this doesn't reduce the number of operations, much fewer weights need to be stored in memory. Third, convolutional layers are equivariant to translations. Regarding 2D images this means that a shift of the ifmap in a spatial dimension results in an equivalent shift in the ofmap i.e.:

$$conv(ifmap(x - i, y - j), kernel) = ofmap(x - i, y - j) \tag{2.7}$$

This property enables CNNs to detect features regardless of where exactly in the spatial input they are located.

Especially the first two properties greatly help with implementing CNN hardware accelerators, since memory requirements are often a bottleneck in such hardware. Hence, historically CNNs have been one of the first ANN layer architectures to be feasible in practical applications [8].

Atrous Convolution

In a standard CNN downsampling indirectly enlarges the field of view (FOV) of later layers and hence allows the CNN to more easily capture features at different scales. However, since through the downsampling operations spatial information is lost, tasks such as semantic image segmentation can suffer from it. *Atrous* convolution, also called *dilated* convolution, is a convolution type with a special kernel structure that increases the kernels FOV. This comes with the benefit that differently sized features can be extracted without further downsampling. Therefore, atrous convolution has been deployed with tremendous success in different fields (e.g. in image segmentation [16, 33, 34] and audio processing [35]).

Essentially, a conventional convolution kernel is padded with zeros between its weights according to the so called atrous-$rate$. Atrous convolution with $rate = 1$ corresponds to the kernel without any padded zeros. Atrous kernels at different rates are shown in Figure 2.4.

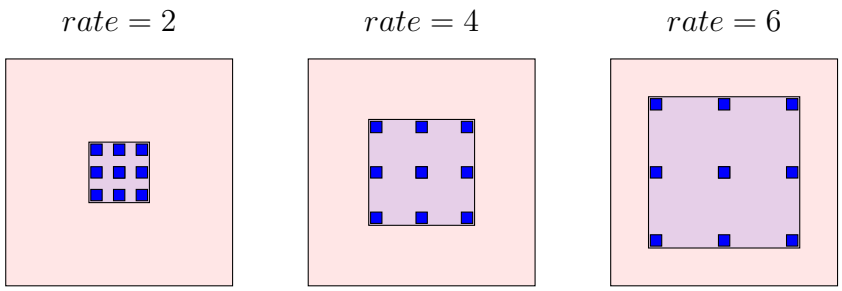

Figure 2.4: Different sized atrous convolution kernels. The kernel is shown over the ifmap with the ifmap shown in red. The kernels are shown in blue, with the non-zero elements highlighted. Figure adapted from [16].

While an atrous kernel can learn larger sized features (in a 2D context) without the need for more parameters than a conventional convolution kernel, the efficient processing poses some challenges due to the additional zero padding introduced between the weight values.

The zero padding between the kernel elements leads to two problems on any computing platform that should be addressed for an efficient implementation: First, only multiplications of non-padded weights with the ifmap should be computed (i.e. the multiplications of the iacts with the non-highlighted blue kernel regions in Figure 2.4 should not be executed since the results are known to be zero). Second, due to comparatively large atrous rates being utilized (e.g. $rate = 12$ on an ifmap with width and height equal to 33) and the same-padding being applied, the multiplication of weights with padded ifmap values are frequent. Since the results of these multiplications are known a-priori to be zero these multiplication operations should also be omitted. Moreover, the frequency of multiplications with padded zeros increases for larger

atrous rates. This can be seen in Figure 2.5, the amount of valid weights drastically decreases for large atrous rates and hence the amount of valid multiplications also drops.

In this thesis the two problems discussed above are dealt with by splitting the filters into multiple 1×1 filters and employing an adapted version of the row stationary dataflow. This is further discussed in Chapter 4.2.

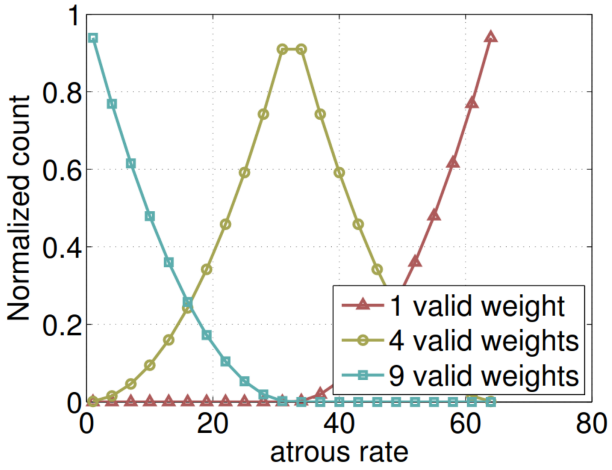

Figure 2.5: Normalized counts of valid weights with an atrous convolution kernel of size 3×3 on a 65×65 ifmap. The figure shows how many weights of the kernel are valid (i.e. not convolving with the padding) vs. the atrous rate. Figure taken from [16].

2.3 DeepLabv3

DeepLabv3 [16] is a specialized neural network architecture employed to do semantic image segmentation. It has achieved very good results on a variety of image segmentation benchmarks, e.g. the 'PASCAL VOC 2012' dataset. Example labelled training data from the PASCAL VOC 2012 (augmented) dataset is shown in Figure 2.6.

The architecture of DeepLabv3 as presented in [16] consists of a backbone and a head network. The backbone is either ResNet-50 or ResNet-101 - in either case a residual network comprised of a total of 50 or 101 layers respectively. These layers are organized into blocks as per the ResNet architecture. Due to the stride operation being employed in ResNet the ifmaps are increasingly reduced in the width and height dimension.

To retain accuracy, but also limit the negative effect of the stride operation, DeepLabv3 employs atrous convolution in later blocks of the ResNet architecture and does not further downsample with stride in those layers.

The atrous spatial pyramid pooling (ASPP) head network is applied to the ofmap of the last backbone layer. The head network consists of three parallel 3×3 atrous convolutions with $rates = (6, 12, 18)$, one 1×1 convolution and a global average pooling module with a subsequent 1×1 convolution. After processing of the convolutions batch normalization and a

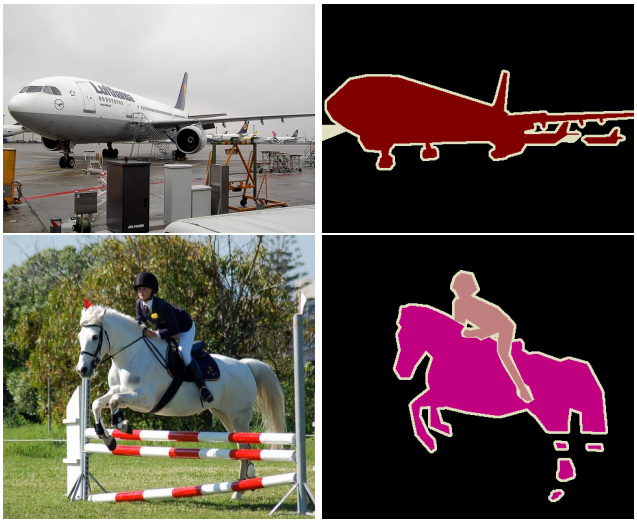

Figure 2.6: Labelled data from the PASCAL VOC 2012 (augmented) dataset [36]. The input images (left side) are shown vs. their respective labelled ground truth (right side).

ReLU layer are applied. The ofmaps of the parallel convolutions are concatenated and passed through additional layers before the final logits are computed. Figure 2.7 shows the architecture of the head network.

The ifmap of the parallel convolution modules of the head network is comprised of 2048 ifmap channels, these compute 1280 ofmap channels through the parallel convolution module. Hence, each parallel module has a filter size corresponding to 2048 ifmap channels and 256 ofmap channels. Therefore, the 3×3 atrous layers have filters of dimensions (2048, 256, 3, 3) and the 1×1 layer in the parallel module has dimensions (2048, 256, 1, 1).

The global average pooling module computes the mean of each ifmap channel and subsequently applies a 1×1 convolution with 256 ofmap channels.

The results shown in Chapter 5 were generated with a reduced version of the atrous convolution layers from the head-network presented in this chapter.

2.4 Hardware Acceleration Techniques

While today neural networks are a popular solution in many applications due to their unrivalled performance in many fields, they are also very computationally expensive. While this can be an acceptable nuisance, if the inference is done in the cloud, it becomes a major problem if inference must take place on the edge. The reasons for requiring inference on the edge, and more specifically embedded devices, are manifold and range from security and privacy considerations to tight real-time and latency requirements [19]. While GPUs can be utilized to do inference on the edge, they are power hungry and cannot fully utilize recent algorithmic advances (as discussed later in this section). Therefore, ASICs and FPGAs are often used as neural network accelerators for embedded devices (e.g. [21, 37]).

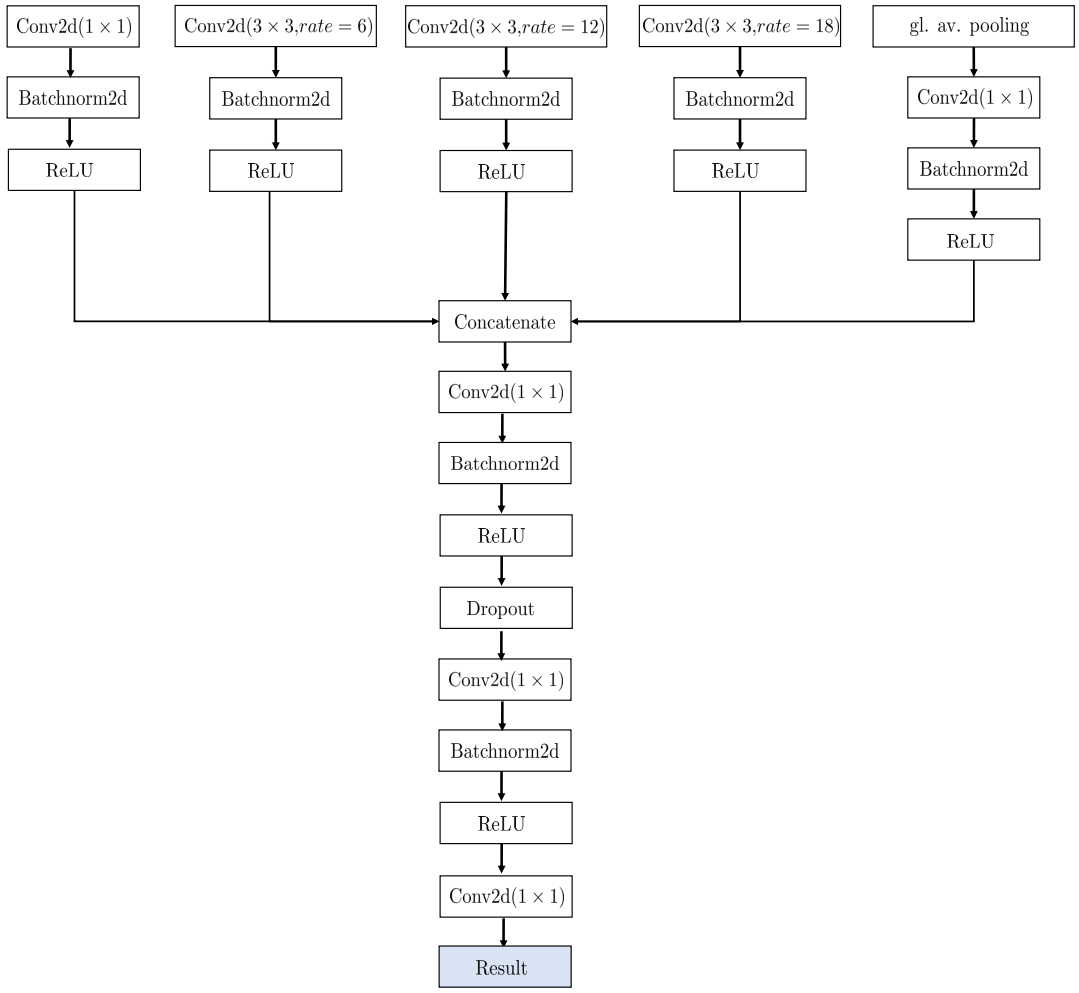

Figure 2.7: DeepLabv3 head architecture.

While ASICs offer a higher clock rate, lower power consumption and a lower per-unit production cost, they suffer from a long time to market and high initial costs. The advantages of using FPGAs range from shorter time to market, much lower initial manufacturing cost to the possibility of reconfigurability. Due to these reasons many FPGA-based accelerators are available (e.g.[2, 38, 39]). More recently, FPGA vendors started offering their own solutions with a wide variety of configuration options [22, 23].

Since the training of the neural network can take place in the cloud and must be done seldom in relation to the inference, most accelerators focus on accelerating the inference [19].[2]

When designing an accelerator, a trade-off between the accuracy, throughput, latency, power requirements and potentially other factors regarding the inference needs to be made [17, 19]. In recent years advances in algorithms have led to techniques that reduce power consumption and

[2]Additionally, since inference is an important part in training, a speedup of inference can sometimes directly benefit training [19].

increase inference speed by orders of magnitudes, while keeping the accuracy loses low or even non-existent [19]. The following sections try to provide a short summary of some important acceleration techniques and specifically discusses the acceleration techniques used in this thesis in more detail. For a more comprehensive summary the reader is referred to [17] and [19].

Spatial architectures

In spatial architectures an array of *processing elements* (PEs) is utilized (see Figure 2.8). Each of the PEs generally consists of a small *register file* (RF) containing inputs, outputs, and possibly temporary results. Furthermore, each PE has at least one *multiply-accumulate* (MAC) unit.

Because of the structure imposed by the neural network architecture, tensor and matrix operations can be mapped on the PEs before execution. Accordingly, each PE can be provided with values from a central controller and only very limited local flow control is needed. Therefore, each PE generally comes with little overhead to execute assigned operations [19].

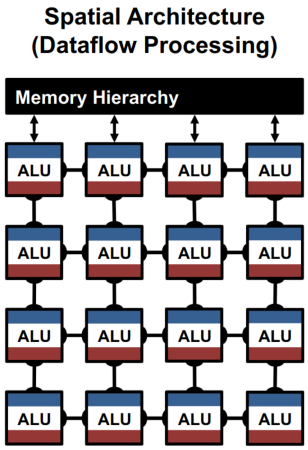

Figure 2.8: A spatial architecture containing a 2D PE array. Figure taken from [17].

Spatial architectures are the defacto standard for neural network accelerators and are routinely employed (e.g. [4, 21, 40]). However, for specific neural networks the spatio-temporal mapping of the computations (*execution mapping*) on the PEs needs to be decided upon which is a critical factor in performance [19]. The accelerator proposed in this thesis uses a spatial architecture. In Chapter 4.2 the mapping of the operations on the PEs is described.

Pruning

Current top-performing neural network models contain a lot of data-redundancy and are often severely over-parameterized [41, 42]. Inspired by biology, where the connections between neurons are often sparse, *pruning* can be employed to reduce the model size. Essentially, pruning drops neuron connections, or even whole structures, that contribute little to the final result. One simple way to do this is to remove weights below a threshold and subsequently retrain the network to recover some accuracy [41]. This results in significantly reduced memory and theo-

retical computation requirements. Furthermore, when combined with retraining, accuracy drops (even for aggressive pruning) can be negligible or non-existent [41]. While there exist structured pruning methods that can be utilized by general purpose hardware [43], often more specialized hardware is needed to efficiently perform the resulting sparse matrix multiplications and incur memory savings [19]. For a more comprehensive overview over state-of-the-art in pruning the reader is refereed to [43].

In this thesis unstructured pruning on filters with the L1 norm is performed in the pytorch framework [44]. Therefore, a specified number of weights in the filters that contribute little to the result (i.e. have small weights), are set to zero and are subsequently disregarded, resulting in pruned connections. The accelerator proposed in this thesis can leverage the resulting reduced computational load in many cases as further discussed in Chapter 5. Since the aim of this thesis is restricted to neural network inference, and multiple previous works have shown that pruning can be applied with minimal accuracy loses, retraining was not performed on the neural network used.

Sparsity

Two forms of sparsity can be exploited in neural networks models. First, sparsity due to the structure of the problem is introduced at design time and is known a-priori. With specialized execution methods and dataflows most of this sparsity can be exploited. Examples for such structured sparsity are the zero-padding around the edges of CNN ifmaps and the kernel padding of atrous convolution.

Dynamic sparsity on the other hand is introduced during inference. Dynamic sparsity starts to appear because, depending on the activation function, ofmap values are set to zero after a convolution layer. In the case of DeepLabv3 the usage of the ReLU activation function leads to increasingly high iacts sparsity in later layers [19]. In the later layers of DeepLabv3 (i.e. the head network) an iacts sparsity of 40-60% is prevalent throughout the ifmaps. The exact sparsity depends on the input image.

Last, after pruning (as discussed previously) the filters also exhibit sparsity. While the distribution of non-zero weights is known at design time, it can be hard to completely utilize it in most unstructured pruning cases. Accelerators like [5] try to alleviate this by reordering the weights such that each PE processes roughly an equal number of non-zero weights at each processing step.

Many accelerators that exclusively leverage weight sparsity, ifmap sparsity or both have been proposed [19]. For a comprehensive review of accelerators see [19]. The accelerator proposed in this thesis utilizes both ifmap and (after pruning) filter sparsity. This is done by utilizing a PE design proposed in [5].

While exploiting sparsity in both the weights and iacts comes at the cost of more control logic, the number of valid MACs is greatly reduced, since both operands need to be non-zero to be considered a valid operation.

Quantization

It has been shown that the precision enabled by 32-bit floating-point multiplications in neural network inference is not needed, rather smaller bit precision like 16-bit floating-point can be employed without reducing accuracy [41, 45]. A wide variety of quantizations have been explored for both weights and activations, ranging from the aforementioned 16-bit floating-point [45], fixed point [46], 8-bit integer [47] to even binary weights [48]. While aggressive quantization results in accuracy drops, techniques like quantization aware training can help with reducing or even eliminating them [49].

While modern GPU architectures offer 16-bit floating-point and even 8-bit integer precision [50], custom ASICSs and FPGAs can utilize even smaller bit precision.

Reduced bit precision comes with several advantages [19]: First, it reduces storage and memory access requirements. Second, hardware units for reduced quantization tensors are far more energy efficient and require less area on-chip. Last, less bandwidth is required on-chip, further reducing the required area, and helping with energy efficiency.

In this thesis 8-bit integer static post-training quantization for both activations and weights is utilized. This quantization can be easily applied by utilizing the pytorch framework [49] and yields a good trade-off between accuracy and compression. When applied to the DeepLabv3 network implementation from [51] an accuracy drop of about 2% in the mean-intersection over union metric (IoU) can be recorded without quantization aware retraining. The 8-bit integer quantization in pytorch is done by mapping the original floating-point values to integer representations using the following formula (the weights are quantized to 8-bit signed, the activations to 8-bit unsigned integers) [49]:

$$x_{quant} = round(\frac{x_{fp32}}{s} + zp) \tag{2.8}$$

The scale parameter s scales the original floating-point value (x_{fp32}) and the zero point zp helps with correctly mapping 0s. Therefore, to compute the quantized result for the next layer (pre-activation function), with w_q, i_q, o_p being the quantized weights, ifmap activations and outputs respectively, $zp_{\{w,i,o\}}$ being the corresponding zero points and $s_{\{w,i,o\}}$ their corresponding scales [47]:

$$o_q = zp_o + round(\frac{s_w * s_i}{s_o} conv(w_q - zp_w, i_q - zp_i)) \tag{2.9}$$

To implement this operation in hardware it is possible to convert the original floating-point parameter $S = \frac{s_w * s_i}{s_o}$ to an 32-bit integer multiplication with a subsequent multiplication of $2^{(-n)}$ [47]:

$$S = S_{int32} * 2^{-n} \tag{2.10}$$

Since the 2^{-n} multiplication can be efficiently implemented using shifts and a 32-bit integer multiplication is much more energy and area efficient than a full floating-point operation, this is the preferred approach to floating-point multiplications. Further elaborations on this quantization approach can be found in [47]. Equation 2.9 in conjunction with the improvement from Eq.

2.10 have been implemented in the requantization pipeline (see Chapter 4.6) of the accelerator presented in this thesis.

Memory accesses

Off-chip memory accesses are generally the most expensive operations for neural network accelerators. In 45nm CMOS technology, accessing dynamic-RAM (DRAM) is $\sim 700\times$ more expensive than a 32-bit floating-point add [41]. While multiplications (the main operation performed during inference) are more costly, they are still two orders of magnitude cheaper than memory accesses [17]. Therefore, it is of great importance to reduce accesses to DRAM as much as possible. This can be done by reusing values already loaded into on-chip memory.

Additionally, quantization and pruning can help to reduce off-chip accesses. While quantization directly translates to less memory traffic, pruning needs special handling to yield improvements. To garner pruning efficiency improvements encoding of the 0s is needed such that these values need not be accessed. This can be done in various ways, e.g. run-length encoding or bitmaps. Furthermore, to further reduce memory accesses, techniques like weight-sharing can be employed. For more information on such techniques see [19].

The dataflow employed in this thesis reduces the theoretical accesses to DRAM. However, utilizing DRAM was outside the scope of this thesis. For testing the accelerator DRAM was replaced with block-RAM (BRAM). When extended with DRAM every weight value would only be read once per input image. No special encoding was applied to reduce theoretical off-chip memory accesses due to pruning or weight-sharing.

2.5 Atrous Neural Network Accelerators

Various atrous convolution accelerators have been proposed, ranging from accelerators that only exploit the structured sparsity introduced by atrous convolution like [1, 2, 3] and one accelerator known to the author, that also exploits dynamic sparsity as introduced by zero iacts and weights [4]. However, the accelerators come with some drawbacks and/or cannot naively be adapted to support dynamic sparsity as is one of the goals in this thesis. The reasons for this are discussed shortly below.

The accelerator proposed in [2] was created with the goal of efficiently processing atrous spatial pyramid pooling similar to the DeepLabv3 head network. To improve the efficiency over a naive solution it supports processing multiple atrous rates in parallel. It does this by utilizing a weight stationary dataflow with a 2D sliding window buffer from where the relevant iacts are extracted. All iacts for the different atrous rates are extracted in parallel and multiplied with their corresponding weights before being accumulated. While this approach makes it possible to heavily exploit ifmap reuse, it suffers from the fact that the number of multiplications that need to be performed varies depending on the atrous rate. For small ifmap sizes this variation can be very high. Hence, some PEs (those associated with higher atrous rates) are severely underutilized since less valid multiplications are recorded for higher atrous rates as discussed in Chapter 2.2. It must be noted that in the case study done in [2] the ifmaps width and height is much larger than in the atrous layers of DeepLabv3, hence the impact is smaller. Additionally,

since the FPGA utilized for this thesis can keep the complete ifmap on-chip, the resulting energy savings are comparatively small for exploiting multiple parallel atrous rates. The author of this thesis believes that, at least for the FPGA used in this thesis, the drawback due to the increased number of computations far outweighs the possible savings because of the iacts reuse.

DT-CNN [1] utilizes delay cells to mitigate the effect of the zero padding between the kernels in atrous convolution. Similarly to [2], the weights are held stationary in PEs and iacts are provided for multiplication. The PEs are organized in rows, where different iacts are propagated column wise. Situated between each PE row, a row of delay cells is placed. These delay the propagation of iacts (through shift registers) as long as required by the atrous rate, such that each PE row is processing the iacts needed for the same kernel. This is shown in Figure 2.9. This approach suffers from a high memory overhead for large atrous rates, since a large number of delay cells would be required. This would be further amplified if dynamic sparsity would be naively exploited into the depth dimension as done in this thesis (as explained in Chapter 4.2).

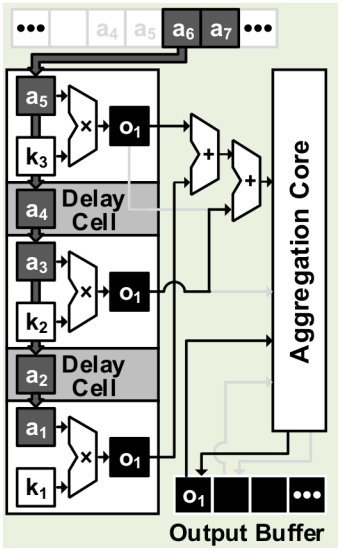

Figure 2.9: In the DT-CNN accelerator the iacts a_i are provided to the PEs which are organized in rows. They are multiplied with their respective weighs k_j and delayed in the delay cells according to the atrous rate, such that the output o_l can be correctly accumulated. The shown example configuration has $rate = 2$ (i.e. one delay cycle). Figure taken from [1].

Both [3] and [4] recognize that reordering of the ifmap in accordance with the atrous rate results in a traditional dense CNN where the structural sparsity introduced by the atrous rate is completely eliminated. While this approach cannot easily benefit from the ifmap reuse as exploited in [2] it enables traditional CNN accelerators to process atrous convolution more efficiently. Furthermore, in [4] the authors propose an architecture that can benefit from dynamic sparsity. However, [4] lacks many implementation details and therefore another accelerator for traditional CNNs, called zero-aware neural network accelerator (ZeNA) [5] was used during this thesis as a reference CNN accelerator. Furthermore, reordering of the ifmap was not needed because of the utilized dataflow.

2.6 FPGAs (ZCU104)

As already discussed, this thesis implements a neural network accelerator on a field-programmable-gate-array (FPGA). To be more precise, a Zynq UltraScale+ MPSoC ZCU104, manufactured by Xilinx [52] was used. It features the Zynq UltraScale+ XCZU7EV MPSoC that combines two Cortex processors (the Cortex®-A53 and Cortex®-R5) and programmable logic. In the course of this thesis only the programmable logic (PL) part of the chip was utilized. The following gives a short overview over the XCZU7EV PL capabilities. For a more in-depth introduction to FPGAs in general the reader is referred to [53].

On an FPGA function blocks are realized with configurable logic blocks (CLBs), which contain look-up tables (LUTs) and flip-flops (FFs). The number of CLBs on the FPGA dictates how many function blocks can be realized i.e. how many PEs can be fitted on the FPGA. More precisely, the LUTs are responsible for implementing logic functions and the FFs are used as register memory. The XCZU7EV has a total of 460,800 FFs and 230,400 LUTs [54].

Another important metric is the on-chip memory size. While large buffers are costly to access energy wise, they still beat off-chip storage. Therefore, a large on-chip size reduces off-chip accesses and makes reusing values easier. There is a total off 38Mb in SRAM on-chip memory. This is split into 11Mb block-RAM (BRAM) and 27Mb ultra-RAM (URAM) [54]. However, this still is only a fraction of the required storage needed for the parameters of the quantized DeepLabv3 which needs more than 400Mb in storage. Hence, using the DRAM is unavoidable if deployment of the complete quantized DeepLabv3 network is desired.

CHAPTER 3

Problem Statement

While neural networks have conquered many fields because of their often unmatched performance and their ease of deployment, they often cannot be run efficiently enough on general purpose hardware. Special hardware neural network accelerators are therefore needed since inference would else be infeasible for many applications.

While there already exist many accelerators for CNNs [5, 21] and some for efficient processing of atrous convolution [1, 2, 3, 4], the accelerator from [4] is the only one known to the author that supports both atrous convolution and dynamic sparsity. However, the publication of [4] lacks many implementation details. Therefore, the goal of this thesis is to implement and test an efficient atrous convolution accelerator that also supports dynamic sparsity. This is done by adapting the CNN accelerator from [5]. Moreover, the proposed accelerator is developed with the atrous spatial pyramid pooling head network from [16] in mind. Furthermore, the accelerator is implemented in the Very High Speed Integrated Circuit Hardware Description Language (VHDL) on a Zynq UltraScale+ MPSoC ZCU104 Xilinx development board. The execution mapping was designed with this specific FPGA in mind. Summing up, the following goals were set for this thesis:

- Develop and use an atrous convolution accelerator dataflow that can efficiently deal with varying atrous rates.
- Improve the efficiency of the accelerator through exploitation of dynamic sparsity as done by PEs proposed in [5].
- Further improve the accelerator by incorporating techniques such as pruning and quantization.
- Develop an accelerator that can be easily scaled to a higher number of PEs.
- Implement the above in VHDL on a ZCU104 Xilinx FPGA.

- While outside of the main scope of this thesis, plan for the accelerator to run the complete Deeplabv3 on the ZCU104 FPGA.

As discussed in Chapter 7, while the presented accelerator leaves room for improvement, the goals where mainly attained.

CHAPTER 4

Accelerator Architecture

This chapter explains the accelerator architecture. First, an overall overview of the accelerator architecture is given. Next the utilized 1×1 - row stationary dataflow is discussed in detail. In the subsequent sections the individual components of the accelerator are discussed.

4.1 Overview

A spatial architecture like ZeNA [5] is utilized, hence the accelerator consists of a 2D PE array. PEs are provided with values for MAC operations over a shared bus by the *control unit* (Chapter 4.3). Values are broadcast row and column wise to the PEs. This reduces *network-on-chip* (NOC) size and memory throughput requirements, since values can be used by multiple PEs. Such a multicast configuration is used by several other accelerators [5, 17].

Each of the rows in the PE array processes the same weights and each column performs computations with the same iacts. This results in each row of the PE array processing a different ofmap channel and each column a different spatial ifmap (and hence also ofmap) position. The full dataflow is discussed in Chapter 4.2.

To reduce ineffectual computations (i.e. multiplications with zero) the values provided to the PEs are passed through the bitvector (bitvec) generation unit (Chapter 4.4) marking non-zero values. These bitvecs are subsequently provided to the PEs for skipping ineffectual multiplications. As discussed in Chapter 2.4 the iacts and weights are represented as 8-bit unsigned and signed integers respectively.

While PEs compute new *partial sums* (psums), the previously computed psums are written out to the control unit, stored, and replaced with older, previously computed, psums to use in the next accumulation cycle. Subsequently, once fully accumulated, psums are sent to the requantization unit (Chapter 4.6) where they are requantized to 8-bit values and stored for final output.

Apart from the quantization, this is also how ZeNA operates. The only major difference to that architecture is that each PE only has a single double buffered psum register in this thesis.

In the current implementation weight and iact values are loaded from on-chip BRAM. For a complete demonstration accelerator that would support the complete DeepLabv3 network, or even only the head-network, it would be necessary that the iacts are loaded by other means (e.g. over UART, for a real accelerator this is infeasible due to the low data rate of UART). Furthermore, the weights would need to be stored in DRAM. Currently they are also stored in BRAM, however on-chip BRAM does not offer enough memory to store all weights. The architecture implemented in this thesis is shown in Figure 4.1.

Over the course of this thesis it became apparent that the approach of only utilizing the PL side of the ZCU104 was suboptimal since the flexibility of the PS side was lost. Chapter 6 proposes a slightly altered architecture that would incorporate the PS. Chapter 6 also highlights the overall place of the accelerator within an application. Certain adjustments that need to be made to implement this are discussed in that chapter, on the other hand this chapter focuses on the implemented parts for a PL side only accelerator.

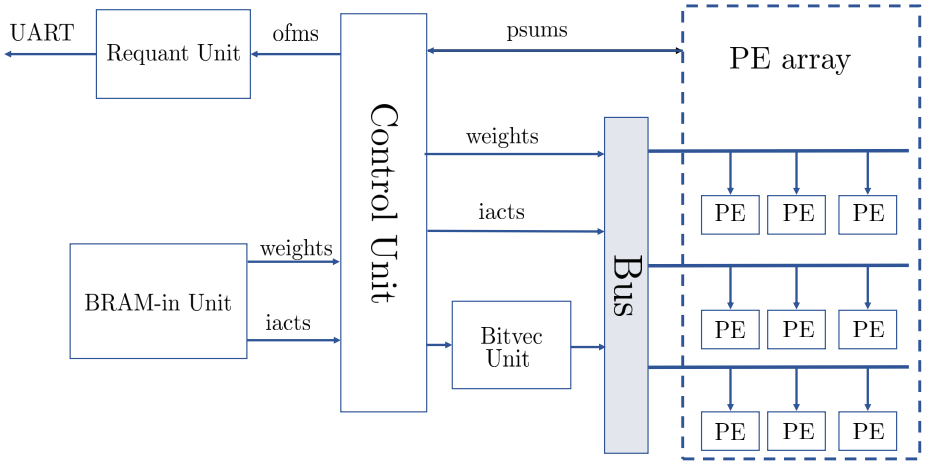

Figure 4.1: Overall architecture of the main accelerator, heavily inspired by [5].

4.2 Dataflow

This section describes the dataflow of the accelerator presented in this thesis. Interestingly, as discussed in [17], the *row stationary* (RS) dataflow is most energy efficient at least for the convolution layers in AlexNet and has been employed in many successful accelerators like [21].

The RS dataflow works as follows: each filter is split into multiple 1D 'row'-filters along the height dimension (splitting along the width dimension would yield a similar dataflow). Subse-

quently, the filter row is held stationary in each PE and iacts are provided in a sliding window fashion to compute subsequent psums [17].

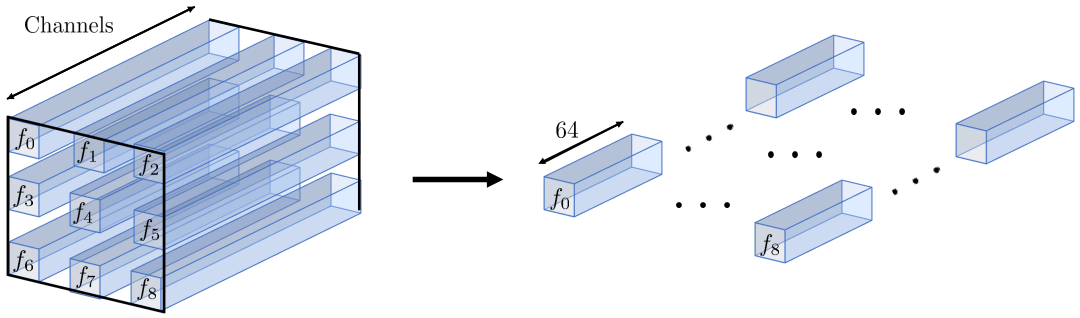

Figure 4.2: One 3×3 filter can be split into multiple 1×1 filters. The filters are first split along the width and height dimension. Furthermore, they have been split along the channel dimension to contain 64 values per filter slice.

The accelerator proposed in this thesis uses a slightly adjusted RS dataflow. Henceforth, it will be called 1×1 - row-stationary (1RS) dataflow. In essence the 1RS dataflow further splits the filters into 1×1 convolution filters along the width dimension. Figure 4.2 shows the splitting of one filter, it must be noted that the filters are further divided up in batches of 64 channels as shown in the figure (this is the number of values each PE receives every *processing cycle*). Consequently, the position of one such filter slice, the *filter position*, can be described in the following way: the spatial location in the original filter ($f_0 - f_8$) and the position according to the depth along the channel dimension of the first value in the slice (fd_d).

Each row of the PE array processes differing ofmap channels and hence receives values from different filters at the same filter position. The filter slices are held stationary in the PEs while relevant iact values are provided to the PEs iteratively. The iacts are distributed column wise over the PE array and vary in respect to their spatial location in the width dimension of the ifmap, i.e. column wise slices of iacts with increasing positions in the x-dimension are distributed. This is visualized in Figure 4.3. Assuming P is increased every time the filter slice has been completely processed by all rows; than the current filter position is described by Eqs. 4.1 and 4.2:

$$f_{(P \mod 9)} \tag{4.1}$$

$$fd_{\lfloor \frac{P}{9} \rfloor * 64} \tag{4.2}$$

Figure 4.4 visualizes the operations performed by one PE array row. As shown, the PE row first convolves a batch of 64 values from f_0 with the relevant ifmap area (the relevant ifmap section is shown over the complete ifmap in red) to obtain the partial ofmap result for one ofmap

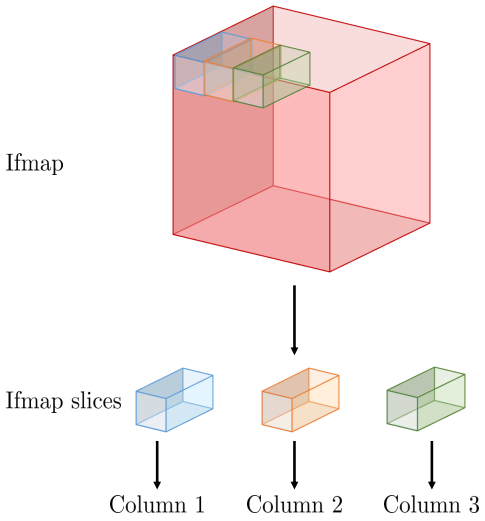

Figure 4.3: Example visualization of the iact slices received by three PE Columns with respect to their original position in the ifmap.

channel (shown in green over the complete ofmap). Next, the other spatial filter positions (f_1 - f_8) are convolved with their respective ifmap sections. Adding up the resulting partial ofmap results yields a partially computed ofmap channel. Once the filter position also reaches the end of the depth dimension the complete ofmap channel has been fully computed. To sum up, each row in the PE array computes its own ofmap channel in parallel. However, each row processes at the same filter and ifmap position.

Algorithm 4.1 presents the described dataflow with one row, one column and one value provided to each PE. The outermost loop reflects the computation of all ofmap channels O. The following loop describes the iterations over all ifmap channels I_D. Furthermore, the kernel splitting into 1×1 convolutions is reflected in line 3 of the pseudocode. Next, since only relevant sections of the ifmap should be convolved with each 1×1 filter the 2D start- and endpoint of the convolution operation on the ifmap are calculated depending on the 1×1 filters spatial position (i.e. the *fil* variable value). In line 8 the psums are accumulated. Last, the spatial position within the channel of the psum depends on the fil variable. This can also be seen in Figure 4.4, where the partial ofmaps are spatially shifted according to the filter position. This position is calculated in line 5.

While Algorithm 4.1 provides the basis for the execution mapping on the PEs, some loop unrolling must be applied to parallelize the execution and provide the PEs with a number of values such that the RF is kept small, and the accelerator can benefit from dynamic sparsity.

To obtain the mapping of the MAC operations on the PE array the loops in Algorithm 4.1 are unrolled as follows: First, to process multiple ofmaps in parallel the **for** loop in line 1 is unrolled

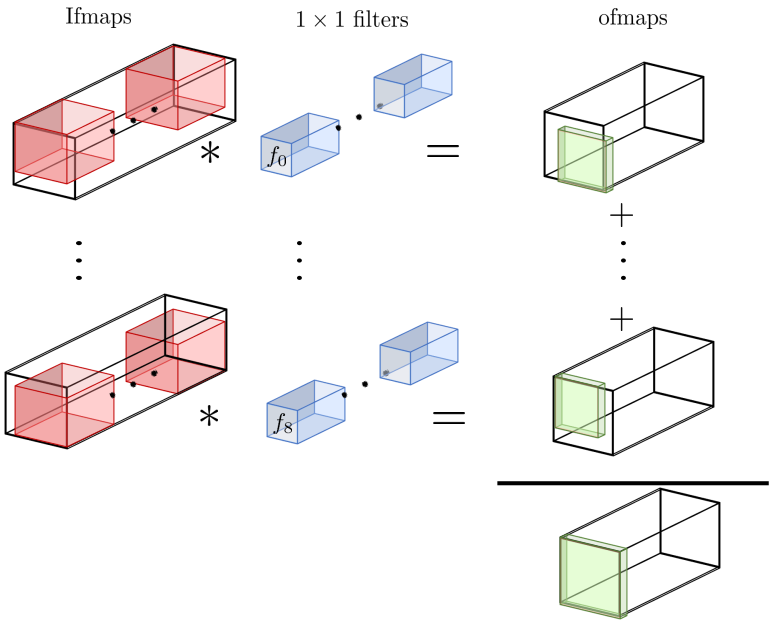

Figure 4.4: Splitting the 3×3 convolution into multiple 1×1 convolutions. The relevant iact section depends on the spatial location of the 1×1 filter in the original filter and is shown in red. The filters are shown in blue. The ofmap that is generated because of the convolution is shown in green on the right side.

in accordance with the wanted number of parallel ofmaps (ofm_p). As previously discussed, these are mapped to different rows in the PE array. To obtain multiple columns, which can benefit from multicasting filters, the **for** loop in line 6 was unrolled. Here the loop unrolling depends on the ifmap size I and the atrous rate r, since if the number of parallel ifmaps (ifm_p) is chosen such that it does not fully divide $I - r$ some PEs cannot be utilized in every processing cycle. Therefore, only values that fully divide $I - r$ were investigated. Additionally, line 7 could be split in the same fashion. This was not done due to reasons outlined in Chapter 5. It is assumed that all currently computed ofmaps are kept in a scratchpad memory. Once they are fully computed they get send to the requant unit and are subsequently stored more permanently. Further remarks and analysis of the impact of different ofm_p and ifm_p values can be found in Chapter 5.

Last, to provide the PEs with batches of values, each PE receives 64 filter values (this is the filter splitting applied along the channel dimension shown in Figure 4.2). To depict this the loop in line 2 is unrolled. Hence, for each processing cycle 64 values are provided to each PE.

As discussed, the implementation in this thesis completely processed one set of 1×1 filters at the same channel depth positions and only subsequently advances the depth. While currently processing all filters of the same channel depth first yields no benefit opposed to advancing into the depth dimension first, the iact values for the current layer depth could be temporarily stored in a smaller buffer in an improved implementation. This would reduce accesses to the large

input : flattened 3x3 filters f with dimensions (O,I_D,9)
input : ifmaps if with dimensions (I_D,$I \times I$)
input : current atrous rate $rate$
output: accumulated psums p with dimensions (O,$I \times I$)

```
1  for (ofm= 0, ofm < O, ofm+= 1) do
2      for (i_d =0, i_d < I_D, i_d += 1) do
3          for (fil= 0, fil < 9, fil+= 1) do
4              start, end= start_end_point_calc(fil,rate);
5              p_o = psums_offs_calc(fil,rate);
6              for (i_x =start.x, i_x < I−end.x, i_x += 1) do
7                  for (i_y =start.y, i_y < I−end.y, i_y += 1) do
8                      p[ofm, i_x+p_o.x, i_y+p_o.y] += if(i_d,i_x,i_y) * f(ofm,i_d,fil)
9                  end
10             end
11         end
12     end
13 end
```

Algorithm 4.1: Dataflow for one PE row and one PE column. The slice size (in the channel dimension) is set to one.

global ifmap buffer by $\times 9$.

The 1RS dataflow was chosen because it completely eliminates the structural sparsity introduced by atrous convolution. First, the added zero padding between valid 1×1 filters can be completely disregarded since 1×1 filters that only consist of zeros can be skipped (the zero only filters are not shown in Figure 4.2). Second, no zero padding at the edges of the ifmap needs to be added since each 1×1 convolution can simply be applied to the relevant subset of the ifmap. Due to the structure of atrous convolution the relevant ifmap subset can be significantly smaller than the whole ifmap (especially for large atrous rates) and therefore large energy and latency savings can be incurred. Additionally, the 1RS dataflow can easily be adapted to other convolution layers.

4.3 Control Unit

The control unit is responsible for implementing the execution mapping on the PEs and providing the PEs with the required values to perform computations. To do so, it implements a state machine. The simplified state machine is shown in Figure 4.5.

The control unit starts in the `LOADING_IFMAPS` state. In this state the iact values are read into the on-chip URAM (in the current implementation the iacts are loaded from the BRAM-in unit). Once the transfer has been finished actual computations can start in the `PROCESSING` super state. Once all processing has been finished the accelerator enters the `FINISHED` state, in which the results are written out over UART.

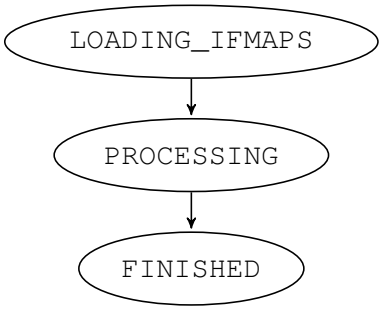

Figure 4.5: States of the control unit.

The PROCESSING state is further split into additional states shown in Figure 4.6. The initial processing state is WAITING. This state is kept until all PEs are ready to receive new values i.e. have finished their current processing cycle and have swapped out their psums. Once this holds the next action is determined. This is either: writing new weight values to the PEs (KERNELS_TO_PES state), writing new iacts (IACTS_TO_PES state) or writing out the current psums to the requantization pipeline (PSUMS_TO_MEM state).

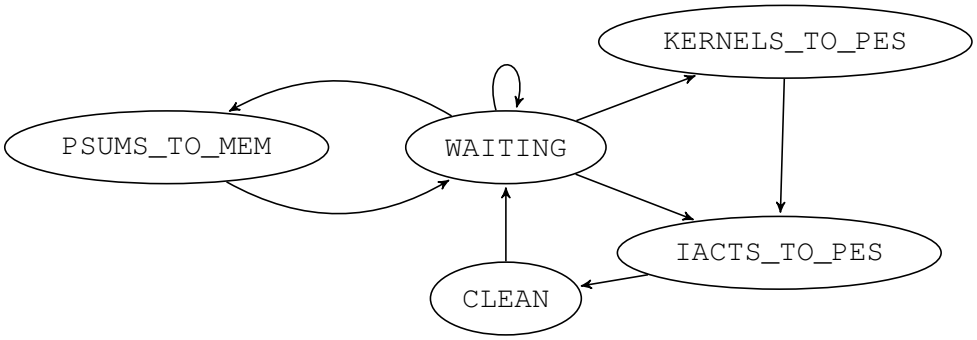

Figure 4.6: State machine of the PROCESSING super-state. The FINISHED state is entered from the WAITING state but was omitted.

If the KERNELS_TO_PES state is entered weights from the BRAM-in unit are requested and subsequently written to the PE array. Since the values are broadcast row-wise, the receiving PE row is encoded in the *new_kernels* std_logic_vector. Next, the IACTS_TO_PES state is invariantly entered, here iacts are written column-wise over the bus to the PEs. As in the KERNELS_TO_PES case the currently selected column is indexed by the *new_ifmaps* std_logic_vector. The PEs start computations as soon as they receive new iacts. Once iacts have been provided to all columns the CLEAN state is entered for one cycle by the control unit. Next, the WAITING state is re-entered. If the current ofmap channels have been fully computed the PSUMS_TO_MEM state is entered next.

The control unit consists of multiple submodules that implement different functionalities. First, the position unit computes the current position in the convolution operation, i.e., it outputs information like the current spatial ifmap and psums position. Additionally, it also outputs

further control information for the control unit to use (e.g. if weight values should be provided to the PE array).

The psums buffer is responsible for buffering the psums, preparing them for the PEs and writing them out to the requantization unit once accumulation has finished. Last, the ifmap buffer stores all iact values and prepares the next needed ones in a smaller buffer. Figure 4.7 shows the control unit architecture. The sub-modules are described in the following chapters.

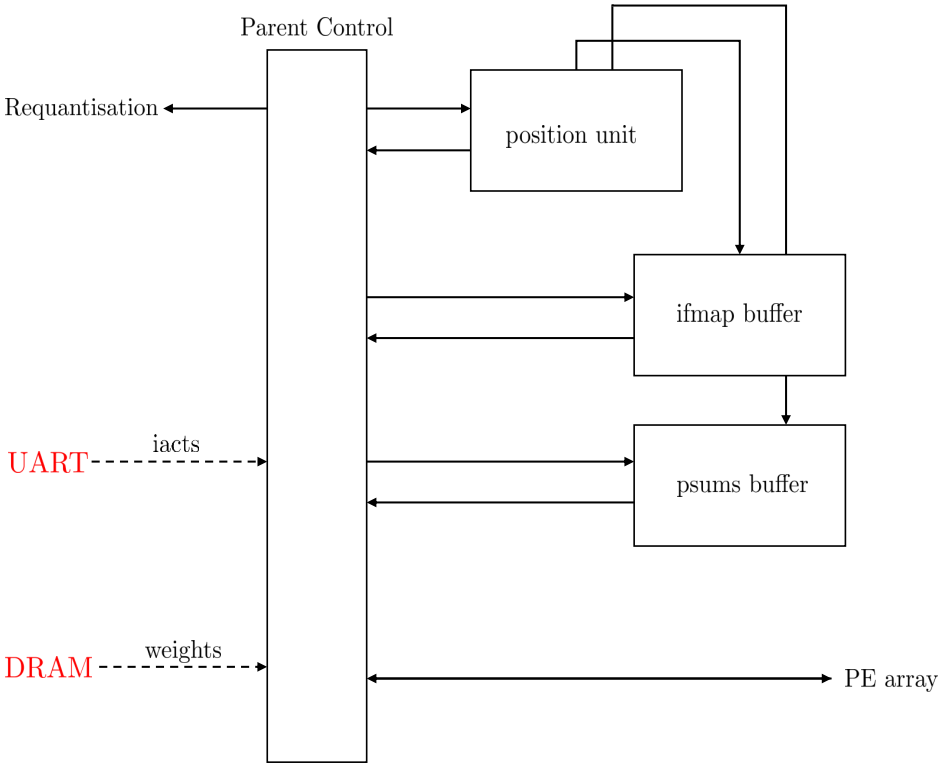

Figure 4.7: The architecture of the control unit. Not implemented interfaces are shown in red. In the current implementation these interfaces are serviced by the BRAM-in unit.

Position Unit

The position unit is responsible for providing the necessary informations for implementing the dataflow from Algorithm 4.1. This is realized by various control signals that notify the parent control module of the current state in the processing of the overall convolution layers. These control signal encompasses for example the *need_kernel* signal, that is used to notify that new weights should be provided to the PEs.

Importantly, the position unit outputs the position of the iacts to be pre-loaded by the ifmap buffer. Furthermore, it computes and outputs the position of the psums that need to be swapped

out to the PE array and the position of the psums that are swapped in from the PE array. From these position values the psums buffer and ifmap buffer derive the actual storage position in their memories. One adaption from the dataflow as discussed in Chapter 4.1 is that the first filter being processed is not the top-left filter (as it would be in a naive implementation) but instead the 1×1 filter in the middle. This was done to efficiently implement the cleaning of the psum buffer after swapping it to the requant pipeline. Since the middle filter iterates over the complete ifmap the cleaning can take place implicitly while doing the filter iteration over the ifmap. Hence, no further special handling is needed.

Ifmap Buffer Unit

The ifmap buffer unit prepares the next iact values that should be provided to the PEs. The iact values are stored in Xilinx Ultra-RAM (URAM). The URAM bus size needs to be chosen as a multiple of 72-bit, however by utilizing both ports of the URAM the size expands to 144-bit. Since the size of the bus to the PE array supports 64 filter values and therefore has a size of 512-bit (without the additional 64-bit in the form of the bitvector), to provide a sufficient number of values per cycle to the PE array the ifmap buffer bus size was chosen as $144 * 4 = 576$-bit. This enables the ifmap buffer to prepare one set of iact values per cycle. Since only 512-bit are required the remaining 64-bit are not utilized. Theoretically, the remaining 64-bit could be used to store the bitvectors such that they wouldn't need to be computed on the fly. This was not done since the bitvector computation is required anyhow for the weight values and the bus size is very specific to the technology of URAM.

Psums Buffer Unit

The psums buffer unit, like the ifmap buffer unit, is controlled by the parent control module. It works only during the `WAITING` phase of the control unit. It performs the following functions: Initially, the psums from the previous processing cycle are requested by the control unit from the PE array. Next, the PE array transfers the psums of the PEs to a temporary buffer of the psums unit. Following that, the content of the buffer is written to the BRAM memory that is utilized by the psums buffer unit. Next, the values that need to be swapped into the PEs are read from the psums memory and are stored in a temporary output buffer. This buffer is subsequently written to the PE array by the parent control module.

The data width of the BRAM memory is determined such that all required psum values for the next cycle have the same address. Therefore, since there are a number of PEs per column (`PE_COLUMNS`), a number of PEs per row (`PE_ROWS`) and the accumulation size per PE is fixed with 24-bit, the data width of the memory (`MEM_DIN`) is equivalent to:

$$\texttt{MEM\_DIN} = \texttt{PE\_COLUMNS} * \texttt{PE\_ROWS} * 24b \tag{4.3}$$

Since the psums memory needs to hold all ofmaps that are currently computed and one ofmap is computed per PE array row, the BRAM has a total size of:

$$\texttt{MEM\_SIZE} = \texttt{PE\_ROWS} * \texttt{OFMAP\_HEIGHT} * \texttt{OFMAP\_WIDTH} * 24b \tag{4.4}$$

Currently, the psums are not written and read over the bus. This should be done in an improved version of the accelerator.

4.4 Bitvec Generation Unit

Values (i.e. both weights and iacts) that are to be broadcast to the PE array must first pass through the bitvec generation unit. In this module non-zero values are marked with a '1' and zero values are denoted with a '0'. Since the values are in a quantized format that utilizes a zero point to correctly depict a zero, the zero point of the current values must be compared with the actual value.

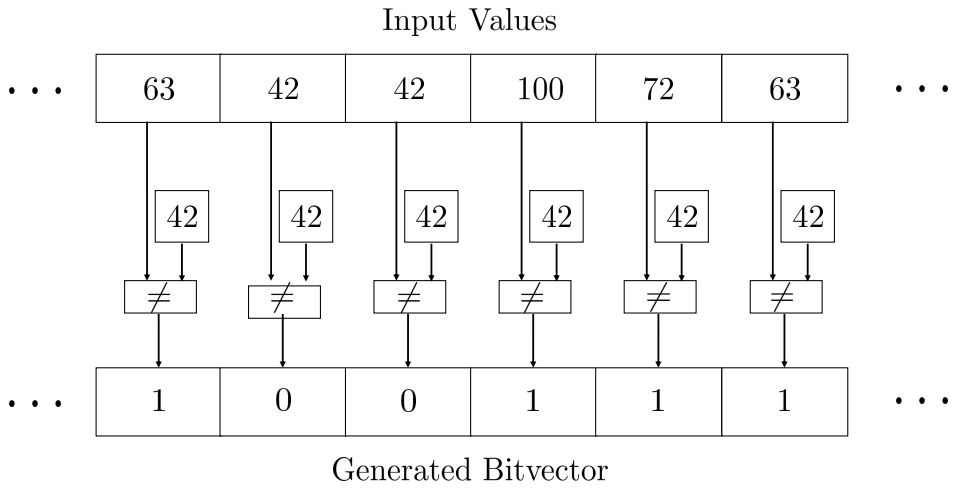

Figure 4.8: Bitvector generation unit, with the zero point for the input values being 42.

Importantly, this zero point varies for weights and the ifmap. However, the zero point of the weights is always zero in the atrous layers of the Deeplabv3 architecture. Figure 4.8 shows an example generation of a bitvec. The module receives input values that are compared with the actual zero point resulting in the bitvec as shown.

Once the bitvec has been computed it is broadcast together with the values over the bus to the PEs. The bitvecs are generated on the fly and are not stored together with the other values, because the overhead from the bitvec generation is insignificantly small in similar accelerators and less memory for storage is required [5]. This is further supported by the resource utilization and power results shown in Chapter 5.

4.5 Processing Element

Each PE consists of three units that together form the PE pipeline. Figure 4.9 shows the slightly simplified PE architecture. When new values are received by the PE over the bus (this is recognized by a high on the *new_ifmaps* signal) the fetch unit starts computing indices of valid multiplications (i.e. multiplications where no participant is zero) and outputs the computed indices to the multiplication unit. In the multiplication unit the weights and iacts values for the multiplications are read from the RF (as indicated by the index from the fetch unit) and multiplied. The resulting value is subsequently sent to the accumulation unit, where it is accumulated until written back to the psums memory of the control unit. This operation procedure allows the PE to skip ineffectual operations. The PE design is heavily inspired by [5]. However, since in [5] a different dataflow is used and not everything is described in complete detail, some adjustments have been made. Mainly the accumulation memory is much smaller since only two psum values are stored.

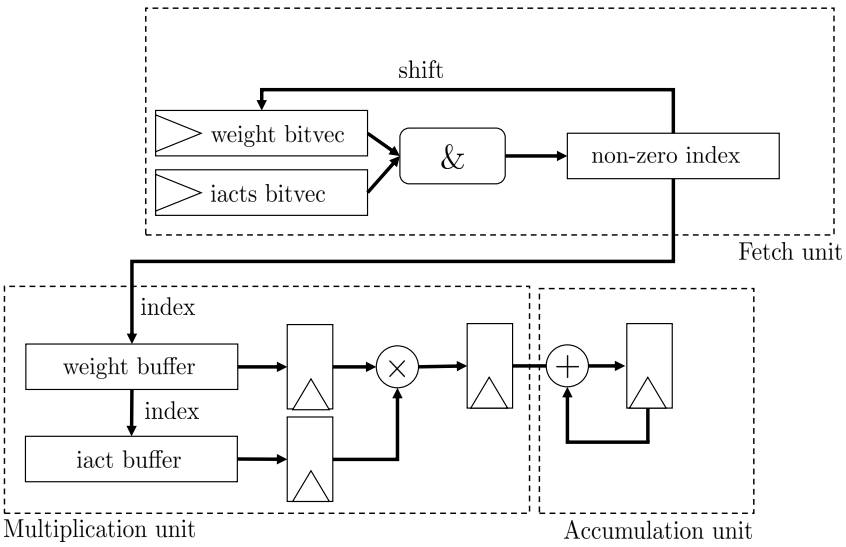

Figure 4.9: Simplified PE architecture, inspired by [5].

Fetch Unit

The fetch unit consists of a small RF where the received bitvecs of the iacts and weights are stored. Furthermore, there are two 16-bit comparison buffers where sub slices of the bitvecs are compared, such that non-zero multiplications are extracted.

The extraction of indices works as follows: a logical AND is applied to the registers in the comparison buffers and a priority encoder returns the first non-zero index. While this index is forwarded to the multiplication unit, the bitvectors in the comparison windows are shifted in

accordance with the extracted index. Consequently, the next indices are computed. Once all indices have been extracted, the PE outputs that it has finished the processing cycle and is ready for new values with a high on the *finished* signal.

The comparison window size of 16-bits is a trade-off since an increase in window size influences the required chip area and power consumption quadratically and limits the achievable clock speed. However, for high sparsity a small comparison window may fail to extract indices and therefore hamper theoretically achievable speed ups of the accelerator. A static size of 16 was chosen because it is comparable to similar architectures (e.g. 24-bit in SNAP [55]). However, by modifying the `COMPARISON_BITVEC_WIDTH` value in the source code different window sizes can be explored.

Multiplication Unit

The multiplication unit has a comparatively large RF where the 64 quantized iact and weight values are stored. This RF is updated if new values are received over the bus. When the module receives a valid index from the fetch unit the relevant values are extracted from the RF and are put into the multiplication pipeline. To yield a correctly accumulated quantized result in accordance with Eq. 2.9 the iact and weight value is subtracted by its corresponding zero point before multiplication takes place. Once computed, the result is extended to 24-bit and sent to the accumulation unit.

Accumulation Unit

The accumulation unit has access to two 24-bit accumulation registers. One of them is the active accumulation register where the current results from the multiplication unit are accumulated. The second register initially stores the results from the last processing cycle. However, during processing this register is read out by the control unit and replaced with a previously calculated psum value. This value will be used in the next processing cycle as the active accumulation register. Therefore, once all value have been accumulated in the current processing cycle (and the non-active register has been updated by the control unit) the non-active register becomes the active one and the active one becomes ready to be swapped out. Apart from the size of these two registers this is also how ZeNA operates [5].

4.6 Requantization Unit

Once a set of psums have been fully accumulated they are written out from the psums buffer unit to the requantization unit for requantization and final storage.

The requantization unit consists of the following components: two BRAM memories that are initialized with the scale ($scale_i$) and shift ($shift_i$) value needed for the requantization (S_{int32} and n respectively from Eq. 2.10), a requantization pipeline where the requantization is realised, a BRAM for final storage of the 8-bit values and additional logic for writing the values from the finalized ofmaps to the UART unit.

The requantization pipeline processes as many values in parallel as are provided by the psums buffer unit in one cycle. Therefore, several values that are equivalent to the number of

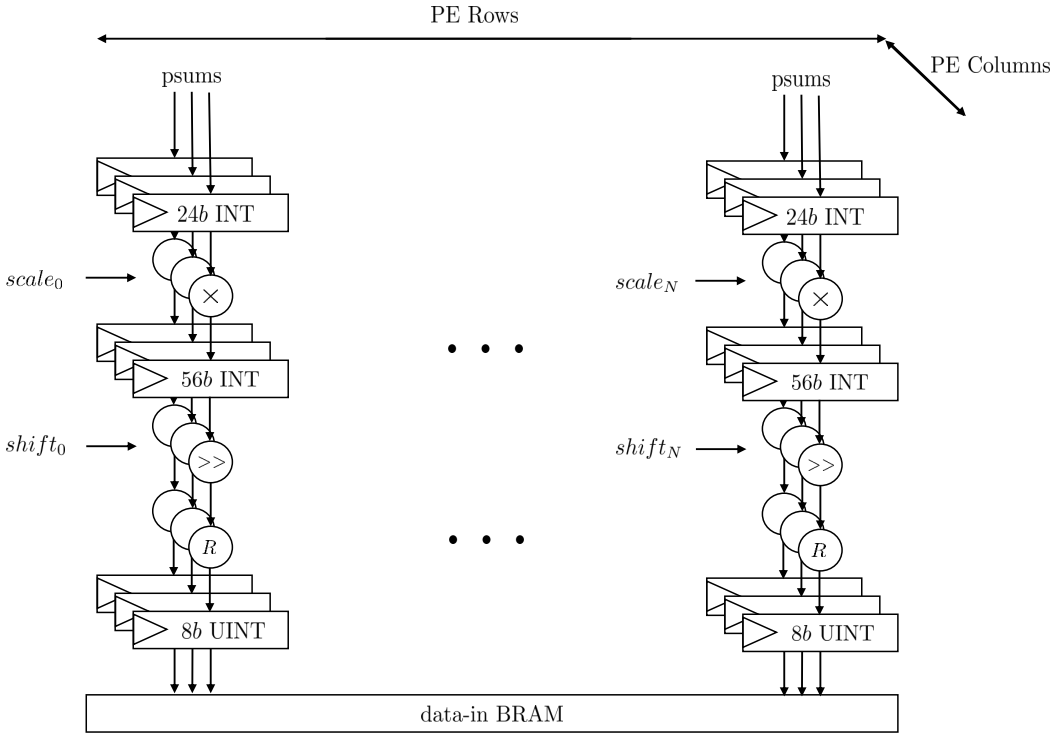

Figure 4.10: The requantization pipeline. Circles denote operations, where $\times$ is a multiplication, $>>$ a right shift and R a round away from zero operation. The widths and datatypes of the registers in the pipeline are shown in the figure.

PEs are processed in parallel in the provided implementation. The requantization pipeline is shown in Figure 4.10. First, the values received are multiplied with their corresponding scale value $scale_i$ in digital signal processing units (DSPs). Since the scale value differs for each ofmap the scale values are stored in several 32-bit registers that are equivalent to the number of PE array rows. After passing through one pipeline register the result is shifted in accordance with the (again ofmap specific) $shift_i$ value. The resulting, shifted value is rounded away from zero, and yields an 8-bit quantized value. The rounding away from zero is done because other rounding operations may introduce accuracy loses as discussed in [47].

The $scale_i$ and $shift_i$ value arrays need to be updated with new values from BRAM during some point of processing on the PE arrays, since the ofmaps are only changed after the completed psums have been fully processed by the requantization unit. While it would be possible to have the requantization pipeline and the PE array process in parallel, this was not done since some additional control logic would be needed (since both the requantization and the PE array require the psums from the psums buffer). Furthermore, the cycles to perform requantization are insignificant with respect to the convolution operations performed in the PE array.

Since the UART unit requires 8-bit values for communicating over the UART interface the requantization unit loads values from the final ofmap BRAM and sends 8-bit values to the UART unit once instructed to do so over the *from_uart* signal.

CHAPTER 5

Results and Discussion

This chapter discusses the results obtained for the accelerator in various configurations. The first section highlights how the results of the following sections were obtained.

5.1 Result Generation

The accelerator was coded in the VHSIC Hardware Description Language (VHDL)[1]. The VHDL code was synthesized, placed & routed in Xilinx Vivado 2020.2. at 100Mhz on a Xilinx ZCU104 development board.

Parameters & Configuration

Since the accelerator does not utilize DRAM, the complete head network of DeepLabv3 could not be deployed on it. Rather, to obtain the results in this chapter, only part of the network was run on the FPGA. However, all the obtained results should directly translate to the complete network. Additionally, the implementation only supports processing and storing of one atrous convolution configuration at a time. After completing a computation, a new configuration needs to be programmed on the FPGA. However, extending this in the future should be straight forward.

As already discussed in Chapter 2 the original dimensions of one 3×3 atrous convolution layer in DeepLabv3 are (1024, 256, `IFMAP_SIZE`). All tests employed a quadratic `IFMAP_SIZE` of 33×33 - corresponding to an original input image with a width and height of 513-527 pixels.[2] Furthermore, the number of ifmap channels and ofmap channels was reduced to 640 and 32, respectively. Therefore, the layer size that was deployed on the FPGA corresponds to (640, 32, 33×33). The original ASPP network from [51] was utilized, however

[1] The code is available on github: [24] and in Appendix B of this thesis.

[2] This size was chosen because it is the default training size in [51].

Adjustable Parameters		
Name	Description	tested value range
PE_COLUMNS	the number of PE columns	1, 3
PARALLEL_OFMS	the number of PE rows	1-32
MAX_OFMS	the total number of ofms to be computed	1-32
DILATION_RATE	the atrous rate	1 (only with PE_COLUMNS=1), 3, 6, 12, 18
IFMAP_SIZE	the width and height of the ifmap	33

Table 5.1: Parameters with which the accelerator can be compiled, the parameters can be set in the *core_pck.vhd* file.

only a subset of channels was used such that the smaller layer size could be achieved. This reduction is meant to enable benchmarking and show the theoretical capabilities of the developed accelerator. When utilizing DRAM for weight storage, the whole layer size would be supported.

The accelerator was compiled with various configurations. Table 5.1 gives an overview and short description of the adjustable parameters.

Software flow & data generation

To check the FPGA computations for correctness and export the filter and ifmap data to the FPGA some scripts where developed in the python programming language. The workflow to run the accelerator is shown in Figure 5.1.

The *deepLabv3main.py* script is adapted from [51]. The script prunes, quantizes the DeepLabv3 network and runs it. During execution the script extracts input ifmaps of the head network during the forward pass. Furthermore, the filter values and requantization parameters are extracted. These values are saved in the binary numpy files *iacts.npy*, *weights.npy* and *outputs.npy*.

Next, the *export_data.py* script can be invoked. While the *deepLabv3main.py* script returns binary files that are filled with data needed to run the complete head network, *export_data.py* can be used to extract a subset of the head network which can be run on the FPGA. Additionally, the script converts the exported subset of the head network into a format that can be used to initialize the BRAM-in unit during synthesis. Furthermore, the *export_data.py* script also calculates the requantization parameters according to Eq. 2.9 to export and later store them in BRAM. Last, the expected result is calculated and stored in the *result.data* file.

With the files generated by the *export_data.py* script Vivado can start synthesis, place & route and generate a bitstream to be uploaded to the FPGA. As soon as the bitstream is uploaded to the device, processing starts. Once processing finishes the FPGA writes its results out over UART. The output was captured with miniterm using the `-raw` keyword (this is important since otherwise not all transactions are recorded successfully). Once the complete output is received over UART, the output data (saved in the file *out.data*) needs to be post-processed. This is done in

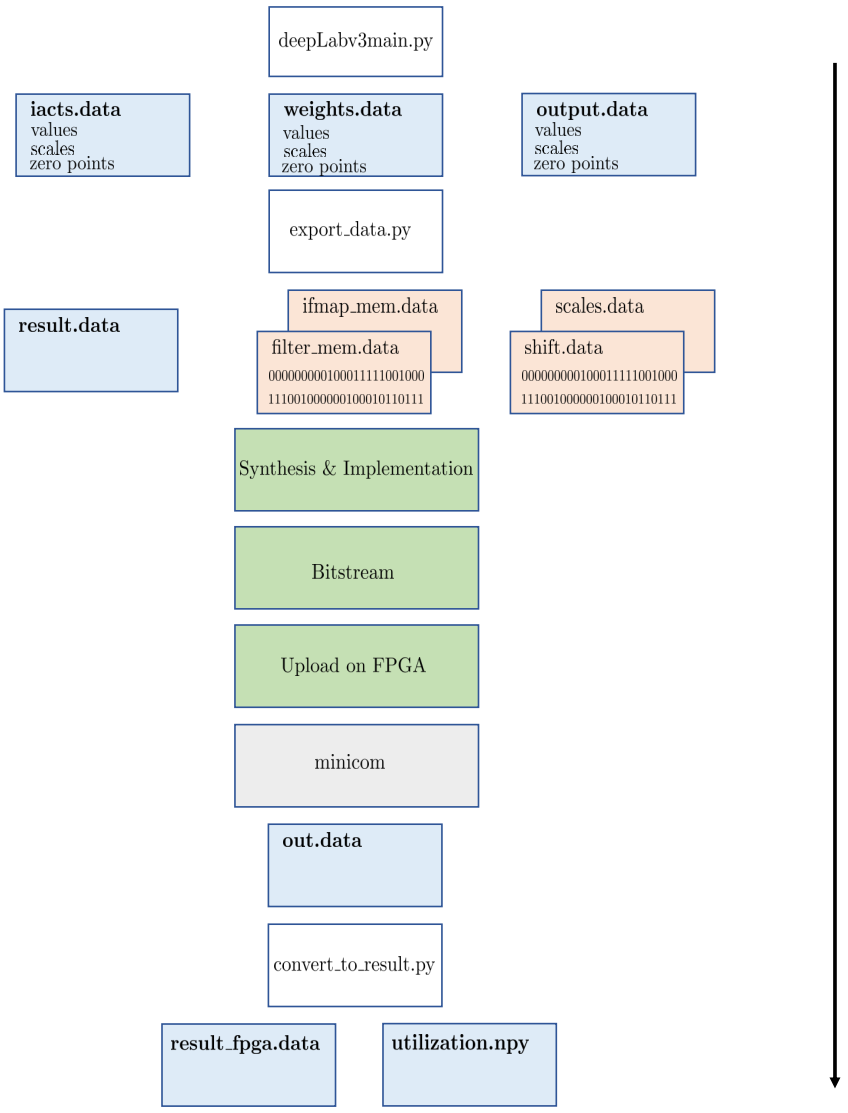

Figure 5.1: Software tools

the *convert_to_result.py* script. In this script the *out.data* file gets read in and device information (like number of total clock cycles ...) gets exported to a numpy binary file (*utilization.npy*). Furthermore, the *result_fpga.data* file is created where the received output feature maps have been converted into the same format as in the *result.data* file. To check if the result received from the FPGA matches a unix-like *diff* off the two files can be done. All plots in the following sections were created with data from the *utilization.npy* file. Further elaborations on how the command line interfaces operate can be found in the `README` of [24] or in Appendix A.

Utilization

The utilization of one PE measures how much effective use is made of it. A low utilization indicates that the PE spends many idle cycles not computing multiplications and therefore might be wasting area and energy. The utilization of an accelerator therefore helps measure how efficiently it can process data. Hence, a high utilization is desirable.

The utilization of the individual PEs was measured with additional logic on the FPGA. Each PE was extended with one additional register that counts the number of valid multiplications that are performed inside the PE. Furthermore, there is a global register that counts the number of clock cycles since processing was started. After processing has finished all utilization count registers are written out over UART and are subsequently post-processed off-FPGA. The utilization u of one PE can be calculated as:

$$u = \frac{\#valid\ multiplications\ per\ PE}{\#total\ cycles} \tag{5.1}$$

The average utilization for all PEs is calculated by adding up the utilization of every PE and dividing the result by the total number of PEs.

5.2 Effect of Pruning on Accuracy

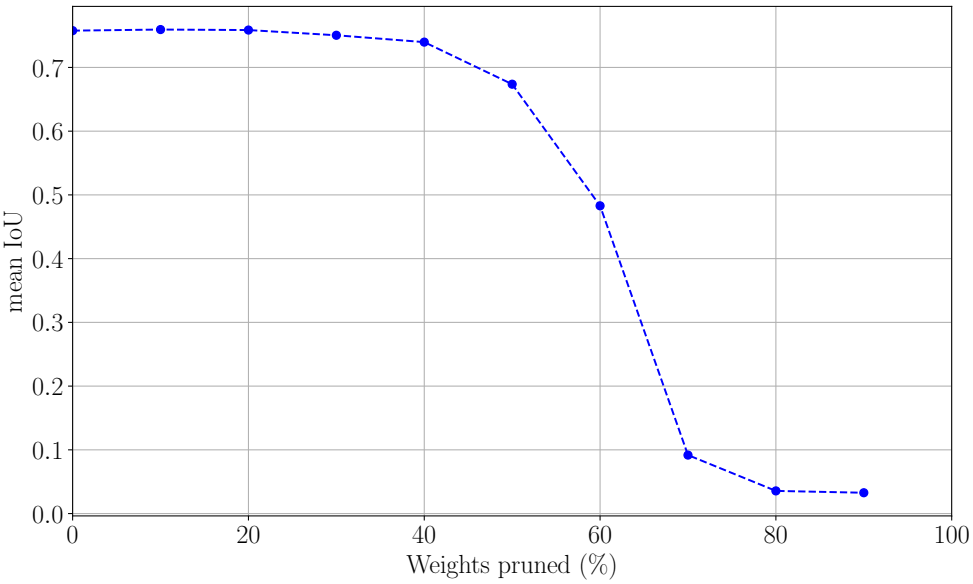

Figure 5.2: Accuracy vs amount of pruned weights in Conv2d layers of Deeplabv3. The weights are pruned for each layer according to the L1 norm. Mean IoU was evaluated on the PASCAL VOC 2012 (augmented) test dataset.

As already discussed in Chapter 2 pruning can be utilized to reduce the amount of valid weights. In order to modify the number of effective multiplications for the results presented in this chapter the pruning rate was varied. Pruning was applied to all convolution layers in the DeepLabv3 network. Figure 5.2 shows the mean IoU of DeepLabv3 vs a varying pruning rate without any retraining on the PASCAL VOC 2012 (augmented) test dataset. As can be seen in the figure, only pruning a low amount of weights results in a very small decrease (or even a slight increase) in the mean IoU accuracy metric. However, at larger pruning rates networks accuracy starts to deteriorate heavily.

5.3 Results & Discussion of the Runtime

This section discusses the obtained results concerning the runtime and utilization of the accelerator. First, scaling of the number of PEs is discussed.

Next, the two main factors that reduce the utilization of the PEs in the accelerator (also influenced by the scaling) are discussed. These factors are PE work imbalance and sparsity of the weights and iacts. Last, the performance of the accelerator in processing atrous convolution is evaluated. All figures shown in this section were generated with an atrous rate of 6 if not noted otherwise.

Scaling the number of PEs

Different number of PEs were synthesized by increasing the number of ofmap channels that were computed in parallel (i.e. increasing `PARALLEL_OFMS`), while keeping the number of `PE_COLUMNS` at three. The only exception in the presented figures in this chapter (if not otherwise noted) is the datapoint at one PE, here only a single PE is used by setting both `PARALLEL_OFMS` and `PE_COLUMNS` to one. As discussed in the previous section the `PE_COLUMNS` correspond to the amount of columns and `PARALLEL_OFMS` to the number of rows in the PE array. Hence, the total number of PEs equals the `PE_COLUMNS` times the `PARALLEL_OFMS`.

The reasoning for increasing the parallelly processed ofmaps instead of the number of columns is twofold. First, if the number of `PE_COLUMNS` does not fully divide $I - rate$ some of the PEs are idle when processing on the edges of the ifmap since all PEs process iact values in the same spatial ifmap region. While an advanced version of the accelerator could be able to cope with this, the current implementation would have to supply the PEs processing outside of the valid ifmap region padded zeros. This restricts the value of `PE_COLUMNS` that were explored to one and three, since otherwise the PE utilization would decrease for at least one of the atrous rates $r = (6, 12, 18)$ present in the head network. Possible optimizations to further scale the number of columns without requiring padding are discussed in Chapter 6.

Furthermore, ifmaps must be supplied to the PEs much more frequently than weights, since weights are only updated once they have been convolved with all their relevant iact values. Since PEs are provided with new iacts once all PEs have finished the previous processing cycle, each PE column receives values from the bus one clock cycle after another. Statistically, this results in the last PE column that receives new values to process longer and hence delay the start of the next processing cycle. The impact of this delay can for example be clearly seen in Figure

5.3 in the *eff_mul: 1.00* data line. Here, the first datapoint where only one PE processes has a utilization of 96.94%; however, the utilization drops by almost 3% down to 94.08% once the PE columns are expanded to three.

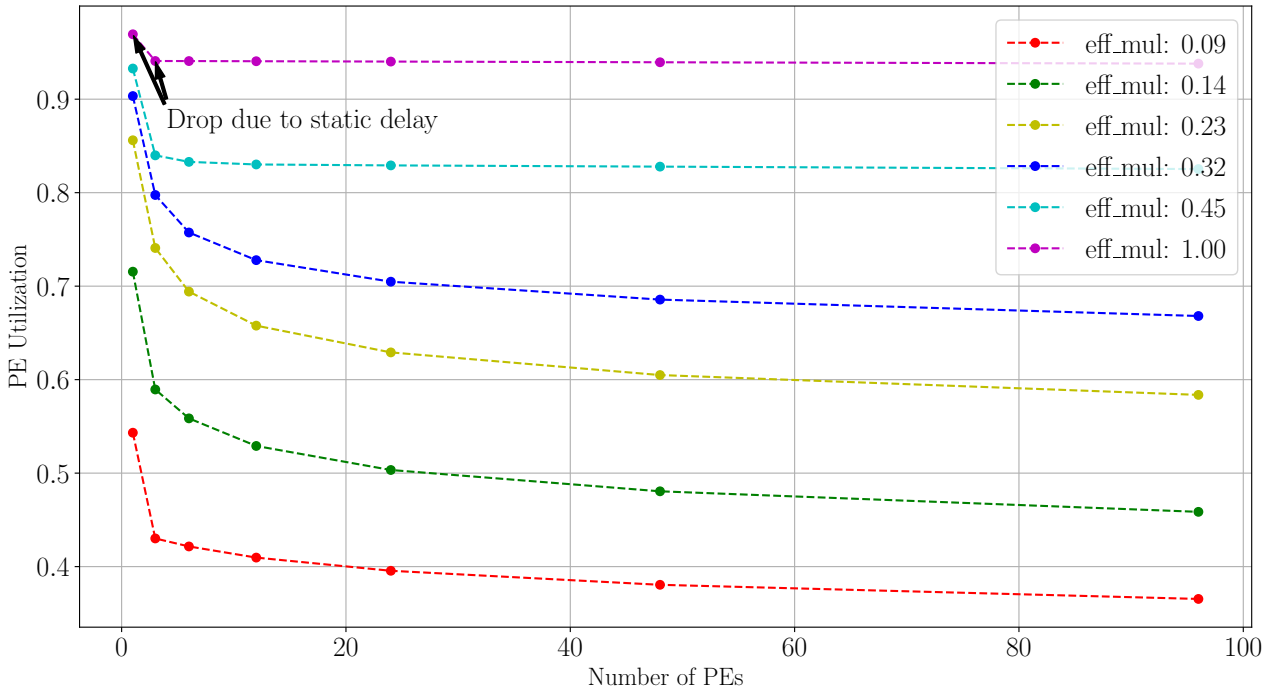

Figure 5.3: Average PE Utilization as a function of the number of PEs for different numbers of effective multiplications. The amount of effective multiplications was varied by adjusting the amount of weights pruned and using different ifmaps. The first datapoint (at one PE) was generated with `PE_COLUMNS` and `PARALLEL_OFMS` set to one. All other datapoints were generated by setting `PE_COLUMNS` to three and varying `PARALLEL_OFMS`.

This drop in utilization is as large as expected. Two setup clock cycles are required as a minimum[3] between a PE reporting it has finished processing and being able to start processing on newly received values. Therefore, the utilization for one single PE in the dense case is $\frac{64}{64+2} = 96.96\%$ in each processing cycle. When expanded to three columns the additional two

[3]For very few valid multiplicatons the number of clock cycles is larger, this is discussed later.

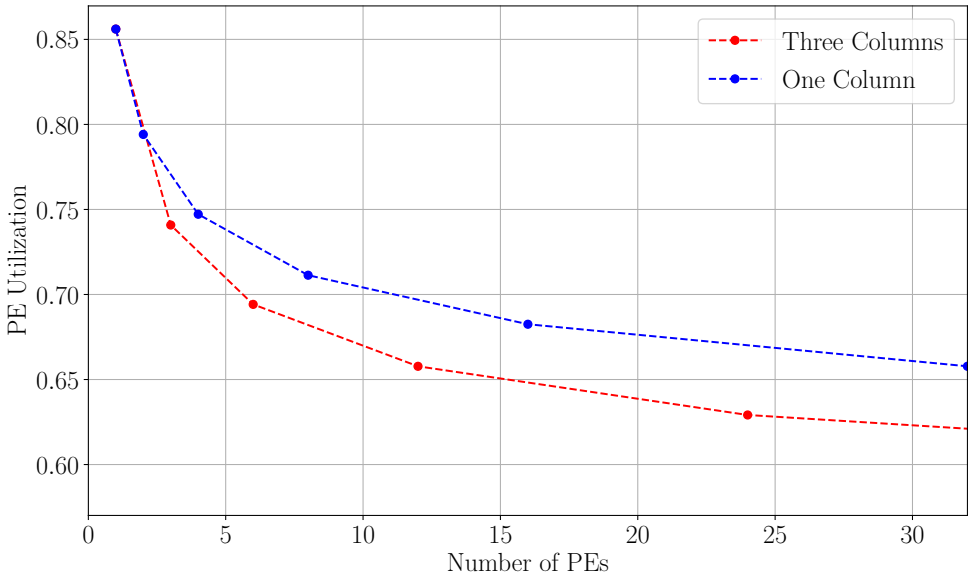

Figure 5.4: The utilization for the accelerator with three PE columns vs. one PE column. It can be clearly seen that the utilization for the one column case is superior. The ifmap and filter values utilized to generate the plot contain around 56% and 50% zeros respectively. The different datapoints were generated by varying `PARALLEL_OFMS` and keeping the `PE_COLUMNS` constant at one and three respectively.

cycles $\frac{64}{64+2+2} = 94.11\%$ introduce the drop by 3%[4].

Possibilities to address the drop in performance both due to the static minimum of two cycles between processing cycles and the drop due to multiple PE columns, both impacting convolution operations with less valid multiplications even more severely, are discussed in Chapter 6.

Because of the above arguments it would be indicated, that utilizing one PE column further improves utilization. While this is the case as can be seen in Figure 5.4, the drawback of computing more ofmap channels in parallel is that the psums buffer needs to be larger, since each currently accumulated ofmap channel needs to be stored in this buffer. As discussed in Chapter 4.3 the psums buffer needs to be expanded by $33 \times 33 * 24$-bit (for the concrete ifmap size) per additional ofmap channel (i.e. 26kb per added ofmap channel). Accordingly, while the psums buffer size grows linearly with the number of ofmap channels it can be reduced by a factor of the number of PE columns. Hence, it was decided to utilize three PE columns to scale

[4]It has to be noted that the calculated utilization does not match exactly with the measured one because of two reasons: First, writing new weights also take cycles that delay further processing, however this happens rarely. Therefore, impacts on the utilization are small. Second, the calculation does not consider that no processing takes place while the psums are written to the requantization pipeline as discussed in Chapter 4.6. Both effects are negligible in comparison to the number of cycles lost due to iacts being written to the PEs. Scaling the accelerator to its intended size will only further decrease the impact of both effects.

the accelerator.

PE work imbalance

PE work imbalance arises because of different iacts and weight values being processed in different PEs. These exhibit differing sparsity rates and therefore result in different execution times for individual PEs. Since the PEs must wait for the slowest PE before receiving the next values, a larger number of PEs decreases the utilization. This can be clearly seen in all figures that plot utilization vs. the number of PEs e.g. in Figure 5.3.

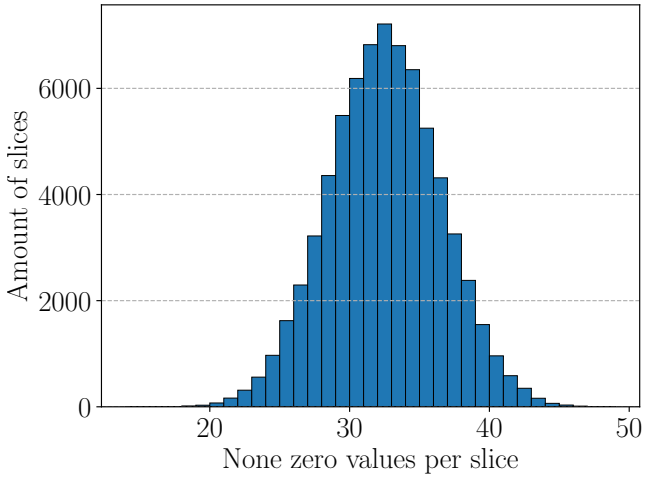

Figure 5.5: Number of non-zero weights in 1×1 filter slices (64 values per slice) that get communicated over the bus. 50% off the weights were pruned.

Interestingly, the number of PEs correlates with the drop in performance logarithmically. This is the case because the number of zero weights follows a normal distribution throughout the 1×1 filter slices received by the PEs. This normal distribution can be seen in Figure 5.5, showing the number of valid weights per filter slice for a specific amount of pruned weights. Furthermore, the number of zeros in the iacts also loosely follows a normal distribution.

To alleviate the problem that the slices provided to different PEs contain differing numbers of non-zero weights, techniques like work stealing and reordering of the weights have been proposed in [5].

In this thesis reordering of the weights was attempted by simply reordering the processing of the ofmap channels. This was attempted because the number of zeros varies significantly between ofmap channels as can be seen in Figure 5.6 (the reordered vs. non-reordered filter are shown). However, only slight improvements (around 1% for 12 PEs) over the non-reorder weights could be obtained.

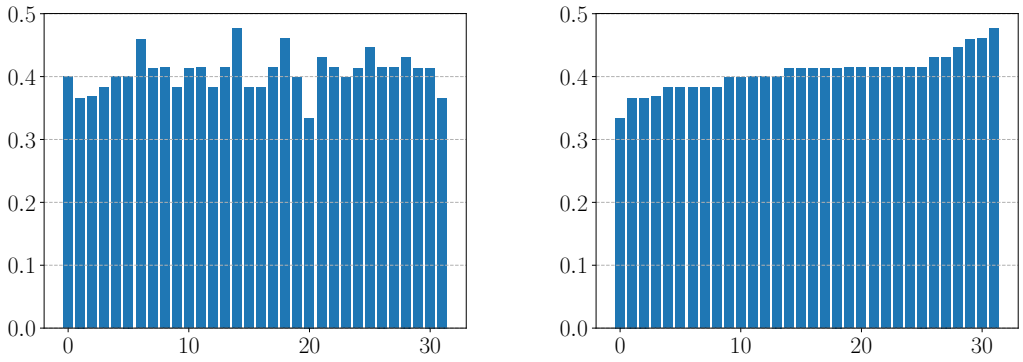

Figure 5.6: Number of non-zero weights in 1×1 filters that get communicated over the bus. 60% off the weights were prunned. The left figure shows the slices before reordering, the right one after reordering.

Effect of sparsity on utilization

While sparsity leads to PE work imbalance, sparsity also has another direct effect on the execution time of each individual PE. There are two main effects that can be observed and decrease the utilization of each PE (and hence also the whole accelerator) for high sparsity. These effects also directly increase the PE work imbalance since PEs processing very sparse values are further delayed.

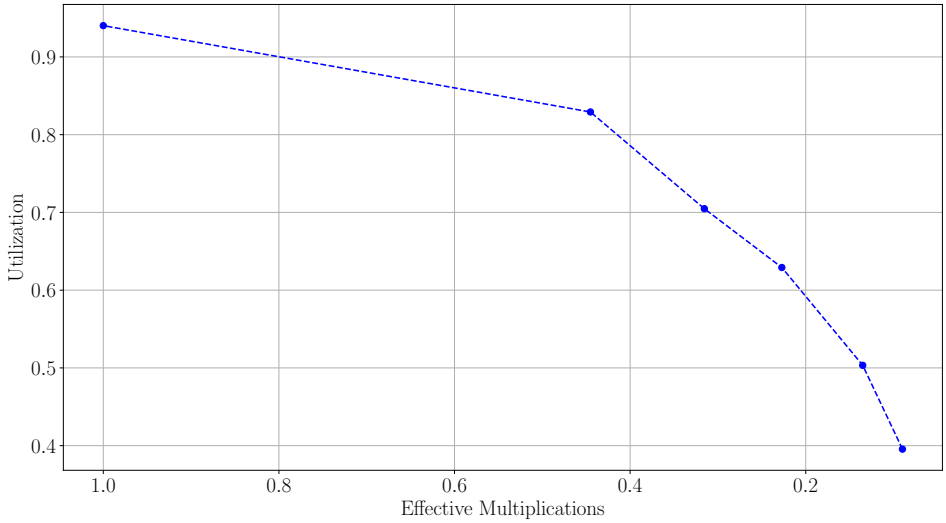

Figure 5.7: Utilization vs. effective multiplications for 96 PEs. The effective multiplications were varied by supplying different ifmaps and varying the pruning rate of the weights.

First, as already discussed, the current implementation has at least two cycles between a PE signalling it has finished processing and starting processing anew with the next values. Since the number of valid multiplications drops with increased sparsity, the impact of the static cycles increases. E.g. for executions where only 10% of multiplications are valid, assuming the PEs can extract one valid multiplication each cycle, the utilization is reduced to only 74% through this effect alone. More so, this effect increases for very high sparsity since the control unit cannot provide the next values fast enough, hence the number of dead cycles increases, further hampering utilization.

Additionally, since the fetch unit in the PEs only has a lookahead distance of 16 the likelihood of not extracting a valid multiplication (and hence an additional dead cycle) rises with an increase in sparsity. The impact of the different effects on the accelerator can be clearly seen in Figure 5.7. In the first part of the figure only a small drop in the utilization can be observed due to the increasing impact of the static cycles. Only later the other two utilization hampering effects (more cycles than two needed to provide values and lookahead) start having a large impact. All the effects are magnified by the scaling of the accelerator and the hence larger varying runtime of individual PEs.

As can be seen in Figure 5.3 the scaling of the number of PEs has a similar effect on different numbers of effective multiplications, implying that scaling the accelerator works sufficiently well and the accelerator is currently mostly limited by effects due to the sparsity. This can be expected, since without the effects due to the sparsity, the utilization drop due to the scaling is limited by the largest difference in values that need to be processed by each PE.

Efficient processing of atrous convolution

Figure 5.8 compares the number of cycles required for different implementation approaches to compute the result of an atrous layer with differing rates. To compare the numbers to the implementation presented in this thesis it was assumed that 96 PEs can be fed with one multiplication per cycle (i.e. the total number of multiplications needed for each approach were divided by 96). The figure only shows the resulting multiplication operations for one specific input image. However, this result should generalize well for other input images with a similar number of zeros. In this case 23% of the multiplications were valid.

The left most bar shows the number of multiplications in a naive implementation where the padded kernel is convolved with the padded ifmap channels and no multiplications are skipped. As can be seen this approach performs poorly and suffers from high atrous rates. The next bar shows an approach where the kernel padding is handled efficiently. This results in a significant drop in required cycles. Additionally, the same performance for all atrous rates is achieved. In [2] such an approach was pursued. The third bar shows an implementation where the kernel and ifmap padding is handled efficiently. This is for example done by [1] and [3]. Through this a further drop in cycles is incurred and large atrous rates require fewer cycles to be processed, coming closer to the ideal case. Furthermore, the red bar shows the number of cycles obtained in this thesis through skipping zeros as described in previous chapters. The last bar shows the ideal case, where every multiplication with zero is skipped and 96 multiplications are executed every cycle.

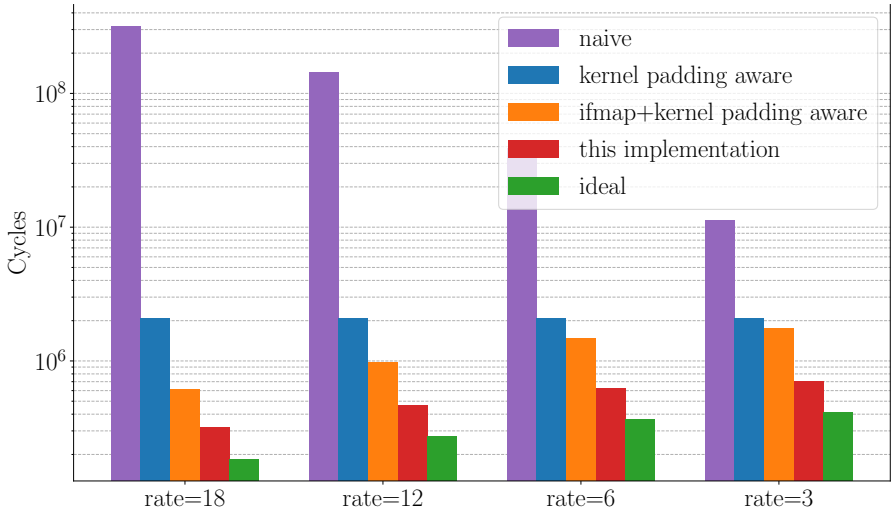

Figure 5.8: Multiplications for 96 PEs with `PE_Columns` $= 3$. The graph has been generated for one ifmap with one set of filters while varying the atrous rate. Around 23% of the multiplications are non-zero to highlight the improved performance of the developed accelerator in the presence of moderate sparsity. Similar figures can be generated for other sparsity rates, however the number of cycles required by the implementation presented in this thesis over the ideal case can also be inferred from the utilization (e.g. reported in Figure 5.3).

It can be observed that the accelerator presented here can leverage the reduced number of operation due to the padding. However, there is still a significant gap between the ideal number of multiplications and the number of multiplications performed by the accelerator. This difference can be explained by the utilization that dictates the size of the gap between the ideal case and the accelerator.

Importantly, since one of the goals of this thesis was to implement an accelerator that performs well for different atrous rates, Figure 5.9 shows that the utilization for different rates stays essentially the same, even when varying the number of PEs. This result is expected since an increased rate only decreases the overall runtime and has little effect on the utilization. The only significant outlier is the case for the large atrous rate 18. However, this outlier can be explained by the fact that the valid ifmap region is reduced quadratically with an increase in atrous rate. Therefore, the distribution of filter weights over the bus also has a quadratically increasing effect on the accelerator utilization. Whereas this effect can be clearly seen in Figure 5.9 it will have a much lower effect on an accelerator processing the complete head network, since weights will be updated much more rarely.

5.4 FPGA Power and Resource Utilization Results

When implementing the accelerator with 96 PEs in Xilinx Vivado 2020.2. on the ZCU104 development board the total consumed power corresponded to 2.329 Watt (1.723-Watt dynamic

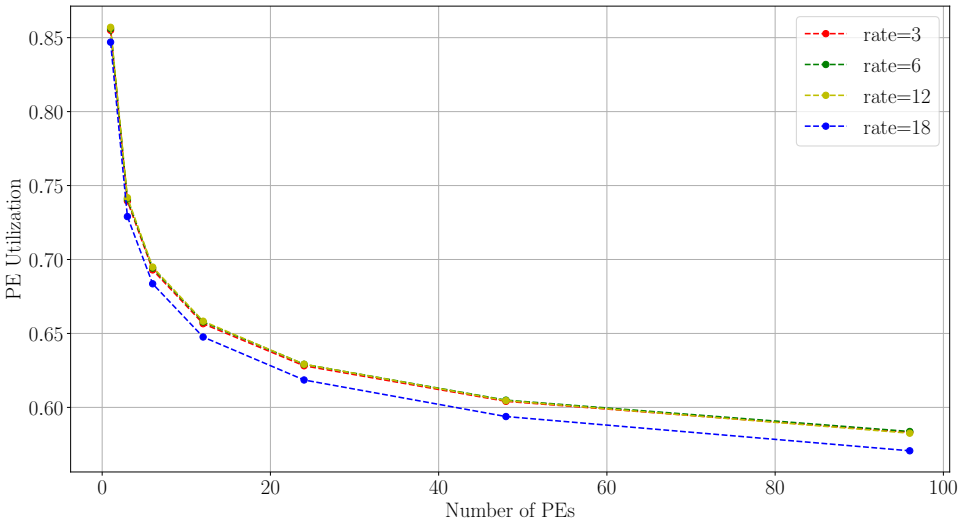

Figure 5.9: Utilization of different atrous rates vs. the number of PEs

and 0.606-Watt static). A further split down of power usage is give in Figure 5.10.

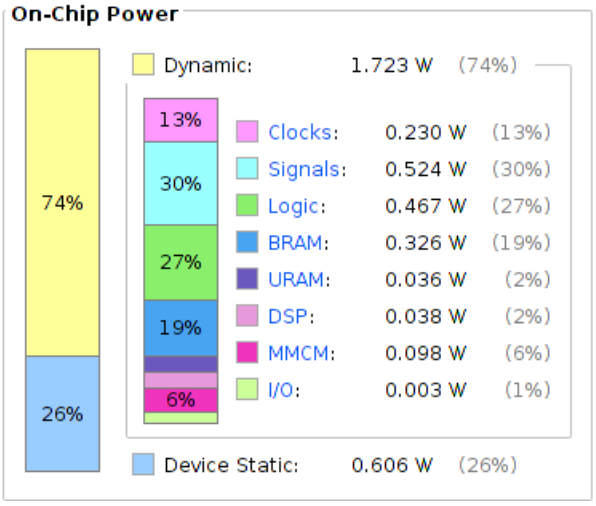

Figure 5.10: Power break down. Generated in Vivado 2020.2.

The only difference to an accelerator running the complete head network should be an increase in power consumption due to an increased URAM and additional consumption due to the DRAM. Since in the current version 33% of the URAM resources are utilized the presented power consumption should be very indicative for an accelerator running the complete head network (except the DRAM).

Table 5.2 breaks the dynamic power down on a per-entity basis. In addition, the resources utilized by the entities are reported.

Module	dynamic Power	LUTs	FF	BRAM	URAM	DSPs
Top	1.723W	122678	106407	257.5	32	193
PE array	0.781 W	89116	96256	0	0	0
PE[1]	0.006 - 0.01 W	$\sim$ 928	$\sim$ 1002	0	0	0
Fetch Unit	0.001 W	0[2]	200	0	0	0
Multiplication Unit	0.002 W	5	$\sim$ 753	0	0	0
Accumulation Unit	<0.001 W	40	49	0	0	0
Leaf Cells	0.007 W	-	-	-	-	-
Bram-in Unit	0.346 W	3031	635	214.5	0	0
Control Unit	0.318 W	5463	6833	32	32	1
Requant Unit	0.073 W	1896	2543	11	0	192
Uart Unit	0.001 W	104	106	0	0	0
Bitvec Unit	<0.001 W	0	1	0	0	0
Clock	0.101 W	-	-	-	-	-
Others	0.103 W	23068	33	-	-	-

[1] Resource usage of the individual PEs differs.
[2] The number of LUTs of the individual PEs does not add up in Vivado 2020.2. It seems likely that the fetch unit should use way more LUTs. This is supported by the fact that Vivado reports a usage of $\sim$ 67 CLBs for the fetch unit (each supporting 8 LUTs and 16 FFs).

Table 5.2: Per entity power and resource break-down.

5.5 Comparison with [2]

Since the accelerator from [2] is also applied to ASPP it seems natural to compare the accelerator proposed in this thesis and the one proposed in [2]. Sestito et.al. report that their accelerator can process an ifmap of size 200×200 with 32 channels and four atrous rates (6,12,18,24) in 0.25ms at a frequency of 181Mhz [2]. The runtime of the accelerator presented in this thesis on the network utilized by [2] can be estimated with Eqs. 5.2, 5.3 and 5.4, assuming that the ifmaps are dense and 100% utilization is achieved[5]:

$$totalMults_{rate} = 32 * ((200 - rate)^2 * 4 + (200 - rate) * 200 * 4 + 200^2) \tag{5.2}$$

$$totalMults = \sum_{rate=6,12,18,24} totalMults_{rate} \tag{5.3}$$

[5] As discussed previously this is unrealistic for the current accelerator, however with some changes proposed in Chapter 6 coming very close to 100% utilization in the dense case is realistic.

$$time = \frac{totalMults}{PEs * frequency} \tag{5.4}$$

Plugging in 96 PEs at a frequency of 100Mhz yields a runtime of 4.334 ms. This runtime is $\times 17$ worse that the one reported in [2]. There are a couple of reasons for the worse runtime of the presented accelerator: First, the presented accelerator was designed for smaller ifmap sizes, hence the atrous rates have a larger effect on the result. When the `IFMAP_SIZE` is reduced to the one investigated in this thesis (33×33) the speedup of [2] is reduced to $\times 10$. Second, the frequency of the presented accelerator was not tuned at all. Third, as apparent in Chapter 5.4 the available on-chip resources of the ZCU104 have not yet been depleted, hence more PEs could be deployed. Last, the effect of sparsity has not been considered.

However, while the accelerator presented in this thesis could probably be made competitive (from a runtime perspective) with [2] through techniques listed above, this comes at a much higher resource usage. This performance difference comes from the fact that [2] utilizes a total of 1152 MAC-units in parallel with little additional overhead. While the here presented accelerator could be scaled a little further, it appears that the overhead introduced due to dynamic sparsity is only worth it in very high sparsity scenarios. Chapter 6.4 discusses how the PEs could be adapted to incorporate multiple MAC units at the cost of removing the dynamic sparsity feature from the accelerator.

CHAPTER 6

Future Improvements

There are various improvements that can be made to the accelerator presented in this thesis. Some important ones are presented in this chapter. Figure 6.1 shows an adapted architecture of the current accelerator, where suggestions presented in this chapter have been included.

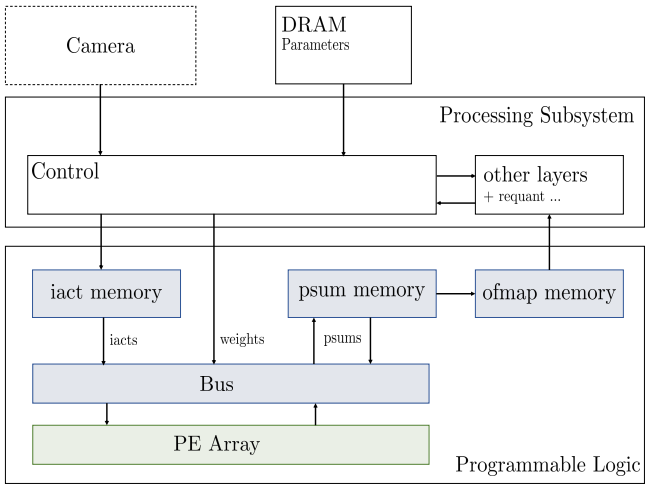

Figure 6.1: Possible complete architecture. Green parts can be used from the current accelerator with minimal changes. Blue modules will need some larger changes.

6.1 Improving Utilization

As discussed in Chapter 5 the average utilization of the PEs dictates the accelerator performance. As already discussed, there are two major reasons for reduced utilization, work imbalance (due

to differing sparsity per PE) and the sparsity itself. This section looks at possibilities to reduce the impact of both.

Work imbalance

First, the work imbalance between PEs can be simplified into an imbalance introduced by varying weight and iact sparsity as done in [5]. The authors in [5] propose to solve each of these problems individually. The imbalance introduced by the weights is reduced by reordering the filters such that each row in the PE accelerator processes a filter slice with a similar number of non-zero activations. In the current state of the accelerator only reordering of the output feature maps is possible. This only increases the utilization marginally as shown in Chapter 5.

However, a more sophisticated filter reordering would be possible. For example, the number of parallelly computed ofmaps could be increased while keeping the number of PE rows the same. Each time weights are communicated to the PEs, filter slices from ofmaps with a similar number of zeros could be scheduled (e.g. instead of 32 parallel ofmaps 64 parallel ofmaps are computed on 32 PE rows, filter slices with a similar number of zeros are communicated together). This would introduce additional complexity, since it would require filter scheduling (currently all the next filters are sent to the PEs) and would require an increase psums memory. While the filter rearranging could happen offline, the state unit would need to inform the psums memory of the current ofmaps to process.

Second, since the iacts also exhibit differing sparsity rates (which in contrast to the filters are not known at compile time), [5] proposes iact stealing between PEs that process the same kernels. This could be added to the accelerator, however since each PE must execute only 64 multiplications at most, it might first be necessary to increase this to yield a real benefit. This could be done in multiple ways.

One possibility would be to increase the number of filters stored per PE. If n filters of the same ifmap channels are chosen (i.e. at the same depth) no additional iacts are needed to compute n psum values. This would increase the average number of multiplications and therefore also the average runtime of each processing cycle by n.

This could be realized in two ways: The additional filters would either need to be part of the currently computed ofmap channels or compute a different ofmap channel. While computing a different ofmap would be easier to incorporate it would come with the requirement of upscaling the psums memory appropriately. The other alternative, namely utilizing multiple filters from the same depth, would in practice mean that multiple spatial 1×1 filters of the same 3×3 filter need to be processed. This can be done with only small additional overhead if the ifmaps are reordered as done in [4].

Alternatively, the RF in the PEs could just be increased in the iacts and filter dimension.

While the solutions proposed above will not completely negate imbalances they could probably increase the accelerator performance in practice. Of course, this comes with the trade-off of additional on-chip logic.

Iact stealing would also enable the accelerator to more efficiently utilize a number of PE columns that does not fully divide $I - rate$. PEs that would be idle due to processing iact values outside of the current ifmap area could assist other PEs by stealing iacts and do processing for them.

Dead cycles between PE processing steps

As already discussed in Chapter 5 there are a minimum of two dead cycles between a PE finishing processing and starting processing on new values. Furthermore, the number of cycles without any work done in the PEs increases even more if more than one column is used. Additionally, if PEs finish processing really fast the control unit cannot even provide new values in two cycles.

An increase in multiplications per PE would directly help with reducing the impact of the dead cycles. Furthermore, double buffering of the iacts, such that new iacts can be preloaded and do not need to be communicated only once all PEs have finished, could further help improve the utilization significantly. While these two extension would potentially be enough to eliminate the decrease in performance due to dead waiting cycles the control unit could be further enhanced to preload the next psums and iacts faster. The current implementation is unnecessarily wasteful in this regard. Improvements in this area alone would help increase the accelerator performance tremendously.

Last, the trade-off of the lookahead in the PEs (see Chapter 4.5) should be further investigated.

6.2 Increasing number of PEs

The scalability of an accelerator is one of its most important characteristics since available hardware resources are steadily increasing. Chapter 5 already discusses how the number of PEs can be scaled naively by increasing the number of rows and columns. While the possible number of columns is limited in the current accelerator, incorporating iact stealing might make other configurations feasible.

Another way to increase the number of PEs per column, without iacts stealing, would be to not only unroll the loop in line 6, but also line 7 in Algorithm 4.1. This would come with the benefit of not requiring additional psums memory, while still increasing the number of PEs per column quadratically.

Furthermore, currently the psums are not communicated over the shared bus. If the accelerator were scaled to incorporate many more PEs, this should definitely be done in order to reduce resource usage.

As discussed in Chapter 5.5 the amount of PEs cannot be scaled easily enough to be competitive with other accelerators. While one possibility is to remove the dynamic sparsity aspect as discussed in Chapter 6.4, another one would be to further explore the implementation of the fetch unit. The resource usage of different implementations should be compared. If a better implementation than the current one can be found, further scaling would be possible.

6.3 Obtaining a full-fledged Accelerator

Currently, the accelerator only supports individual atrous layers from the head network. However, the accelerator can be extended to support the complete DeepLabv3 network. To do this

the accelerator needs to support the remaining operations of the head and ResNet-101 network. Additionally, this would necessitate some changes to the control and memory units.

Layer operations

Currently, the accelerator supports the convolution operation for 3×3 filters, with a variable atrous rate that can be divided by three. Atrous $rate = 1$ and other rates that are not divided by three can only be used with the number of PE columns set to one. This is in part due to the memory layout currently utilized. Table 6.1 gives an overview of operations that are supported and operations that still would need to be implemented for an accelerator that supports the complete DeepLabv3 network.

Backbone (ResNet 101)				
Layer	kernel	stride	rate	supported
Quantization	-	-	-	offline
Conv2d	7×7	2	1	no
Conv2d	3×3	2	1	no
Conv2d	3×3	1	1	yes
Conv2d	1×1	1	1	yes[1]
BatchNorm2d	-	-	-	no
ReLU	-	-	-	no
skip connections	-	-	-	no
MaxPool2d	3	2	1	no
Head				
Layer	kernel	stride	rate	supported
Conv2d	3×3	1	6, 12, 18	yes
Conv2d	1×1	1	1	yes [1]
BatchNorm2d	-	-	-	no
ReLU	-	-	-	no
AdaptiveAvgPool2d	-	-	-	no
Dropout	-	-	-	no
Requantization	-	-	-	yes

[1] None 3×3 kernels only have support for one PE column in the array

Table 6.1: Operations

With some adaptions to the control and psums unit all 2D-convolution operations (Conv2d) can be supported and executed on the PE array. The remaining operations are far less compute intensive and specialized modules on the FPGA to perform them would require little resources. Alternatively, performing these operations on the processor side of the ZCU104 board would be feasible since they require orders of magnitude less compute power.

Furthermore, the batch normalization could be done with the already implemented requantization in tandem, with only changes to the offline computed requantization parameters.

Changes to the control unit

Currently, the control unit is implemented on the PL side of the FPGA. However, for easier further extensions and implementations of some layers in Table 6.1 it would be advantageous to utilize the PS side of the ZCU104 board. Additionally, since a more sophisticated memory management would be beneficial for the full-fledged accelerator (some aspects of this are discussed in the next section), an implementation of the control unit on the PS side would further ease development.

The current control unit can only fully manage a memory layout that supports atrous rates that are multiples of three. This should be changed in a future version of the accelerator for better generalizability.

Changes to the memory

As already stated multiple times, the current accelerator does not utilize DRAM and hence only a limited version of the head network can be run on it. In order to support the complete network DRAM will be needed. Currently, the complete output of one (reduced) layer is saved on chip, this will also be possible for a full-fledged accelerator on the ZCU104 board.

This is the case because the highest memory requirement occurs at the transition from the backbone to the head network. At this point the input layer has dimension (2048, 1280, `IFMAP_SIZE` $\times$ `IFMAP_SIZE`). Therefore, 17.842176 Mbit would be needed for the iact values and 11.15136 Mbit for the ofmap values. Together a total of 28.15 Mbits would be needed in on-chip memory. This leaves about 10 Mbits in on-chip memory for the storage of the psums. This is enough, since only 6.7 Mbits are required when computing all 256 ofmap channels of one atrous rate in parallel and the memory required for storage of other non-weight values (e.g. requantization) is very small. Concluding, the on-chip memory, if efficiently used, is enough to support storage of the complete current ifmap and ofmap. Therefore, no DRAM accesses to swap out ifmap values are required. Instead, they can be kept on chip until deprecated or outputted.

However, the weight values of the model will still need to be saved on DRAM. They will be fetched just-in-time from off-chip since they need a total of about 400 Mbits and therefore on-chip storage is infeasible. Every weight value will need to be read only once, resulting in a low number of DRAM accesses, that can only be further reduced by techniques as discussed in Chapter 2.4.

Last, because the dimensions of the ofmaps changes significantly throughout the network the accelerator needs to be able to deal with this. This implies that a more sophisticated psums and ifmap storing needs to be implemented than is currently present in the accelerator. This, again, would probably be easiest to implement with help of the PS-side of the ZCU104.

6.4 Removing Exploitation of Dynamic Sparsity

As discussed in Chapter 5.5 it appears that the overhead introduced by dynamic sparsity is often not justifiable. However, it also appears as if the presented dataflow performs very well and the decrease in runtime for smaller ifmap sizes is large. Hence, adaption of the PEs might yield

good results. This could be done by simply removing the fetch unit from the PE and utilizing many more multiplication-units (e.g. 32) per PE. The individual results would be added up in an adder tree and written to the accumulation unit. With this change the runtime of each PE is known a-priori, hence some changes to the control unit would also be necessitated. This would allow to scale the number of multiplication units easily. Further investigation of the trade-off between dynamic sparsity and efficient resource usage are needed.

CHAPTER 7

Conclusion

The problem stated in Chapter 3 has been solved: an atrous convolution accelerator supporting dynamic sparsity has been developed on the ZCU104 development board. As shown in Chapter 5, the accelerator can efficiently handle atrous convolution layers:

- The accelerator's utilization is minimally affected by varying the atrous rate.
- The ifmap and kernel padding are efficiently handled by the 1RS dataflow.
- The number of PEs in the accelerator can be scaled with some impact on utilization.
- Sparsity in the weights and iacts can be exploited.

Furthermore, quantization to 8-bits has also been implemented. Additionally, while the current accelerator does not support the complete DeepLabv3, the accelerator was designed with doing so in mind.

However, while the goals laid out for this thesis where attained, there still are some points where improvement is needed to yield a competitive ASPP accelerator. In order to do this the following improvements are needed and should be considered:

- Utilizing DRAM is paramount in order to support the full DeepLabv3.
- The current control unit is somewhat inefficient at low sparsity and should be improved.
- In general, the current accelerator could be improved tremendously with some changes to better handle sparsity, as discussed in Chapter 6.
- Currently, no direct evaluation of resource consumption compared to other accelerators has been performed, but a comparison with [2] suggests that the presented accelerator uses much more resources per PE than other FPGA based neural network accelerators. Accordingly, much fewer PEs can be deployed resulting in the need for very high sparsity rates to outperform an implementation that does not utilize dynamic sparsity. Hence, for

realistic weight and iacts sparsity rates it is not worthwhile to utilize zero skipping at least in the presented implementation on the ZCU104 FPGA platform.

The accelerator proposed here is, at least to the author's knowledge, the first to use the 1RS dataflow described in Chapter 4.2. While this dataflow has some limitations and requires more bandwidth than a traditional row-stationary dataflow, it has the advantage that it can easily handle atrous convolution and other convolution operations.

Overall, the implemented accelerator demonstrates the capabilities of the proposed architecture, but is still somewhat limited in scope.

Bibliography

[1] D. Im, D. Han, S. Choi, S. Kang, and H.-J. Yoo, "Dt-cnn: Dilated and transposed convolution neural network accelerator for real-time image segmentation on mobile devices," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[2] C. Sestito, F. Spagnolo, P. Corsonello, and S. Perri, "An efficient convolution engine based on the à-trous spatial pyramid pooling," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 77–80.

[3] K.-W. Chang and T.-S. Chang, "Efficient accelerator for dilated and transposed convolution with decomposition," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[4] W. Liu, J. Lin, and Z. Wang, "Usca: A unified systolic convolution array architecture for accelerating sparse neural network," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[5] D. Kim, J. Ahn, and S. Yoo, "Zena: Zero-aware neural network accelerator," *IEEE Design & Test*, vol. 35, no. 1, pp. 39–46, 2017.

[6] A. Mayor, *Gods and Robots: Myths, Machines, and Ancient Dreams of Technology*. Princeton University Press, 2018.

[7] M. Tegmark, *Life 3.0*. Penguin Books, 2018.

[8] Y. Bengio, I. Goodfellow, and A. Courville, *Deep learning*. MIT press Massachusetts, USA:, 2017, vol. 1.

[9] R. Szeliski, *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[10] J. Haugeland, *Artificial intelligence : the very idea*, 8th ed., ser. A Bradford book. Cambridge, Mass. [u.a.]: MIT Pr., 2000.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[12] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[13] S. Zheng, J. Lu, H. Zhao, X. Zhu, Z. Luo, Y. Wang, Y. Fu, J. Feng, T. Xiang, P. H. Torr *et al.*, "Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers," *arXiv preprint arXiv:2012.15840*, 2020.

[14] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," 2021.

[15] C. Li, T. Tang, G. Wang, J. Peng, B. Wang, X. Liang, and X. Chang, "Bossnas: Exploring hybrid cnn-transformers with block-wisely self-supervised neural architecture search," *arXiv preprint arXiv:2103.12424*, 2021.

[16] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking Atrous Convolution for Semantic Image Segmentation," *arXiv:1706.05587 [cs]*, Dec. 2017, arXiv: 1706.05587. [Online]. Available: http://arxiv.org/abs/1706.05587

[17] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[18] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of fpga-based neural network accelerator," *arXiv preprint arXiv:1712.08934*, 2017.

[19] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights," *arXiv preprint arXiv:2007.00864*, 2020.

[20] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 5–14.

[21] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[22] "Zynq DPU v3.2, Product Guide," https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_2/pg338-dpu.pdf, accessed: 2021-05-05.

[23] "Intel® FPGA Technology solutions for artificial intelligence (AI)," https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/solutions.html, accessed: 2021-05-05.

[24] " ASPP-accelerator," https://github.com/HyberionBrew/ASPP-accelerator, accessed: 2021-07-26.

[25] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[26] "Drawing Neural Network with tikz," https://tex.stackexchange.com/questions/153957/drawing-neural-network-with-tikz, accessed: 2021-05-14.

[27] F. H.-F. Leung, H.-K. Lam, S.-H. Ling, and P. K.-S. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *IEEE Transactions on Neural networks*, vol. 14, no. 1, pp. 79–88, 2003.

[28] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[29] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 3rd ed. Pearson, 2009.

[30] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[31] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[32] "Drawing Convolution with tikz," https://tex.stackexchange.com/questions/437007/drawing-a-convolution-with-tikz, accessed: 2021-05-14.

[33] Y. Wei, H. Xiao, H. Shi, Z. Jie, J. Feng, and T. S. Huang, "Revisiting dilated convolution: A simple approach for weakly-and semi-supervised semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7268–7277.

[34] L. Zhou, C. Zhang, and M. Wu, "D-linknet: Linknet with pretrained encoder and dilated convolution for high resolution satellite imagery road extraction," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 182–186.

[35] X. Zhang, Y. Zou, and W. Shi, "Dilated convolution neural network with leakyrelu for environmental sound classification," in *2017 22nd International Conference on Digital Signal Processing (DSP)*. IEEE, 2017, pp. 1–5.

[36] M. Everingham, L. Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *Int. J. Comput. Vision*, vol. 88, no. 2, p. 303–338, Jun. 2010. [Online]. Available: https://doi.org/10.1007/s11263-009-0275-4

[37] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky *et al.*, "A scalable multi-teraops deep learning processor core for ai trainina and inference," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 35–36.

[38] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[39] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, "A high-performance cnn processor based on fpga for mobilenets," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 136–143.

[40] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "Snap: A 1.67—21.55 tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos," in *2019 Symposium on VLSI Circuits*. IEEE, 2019, pp. C306–C307.

[41] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[42] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.

[43] J. Liu, S. Tripathi, U. Kurup, and M. Shah, "Pruning algorithms to accelerate convolutional neural networks for edge applications: A survey," *arXiv preprint arXiv:2005.04275*, 2020.

[44] "L1Unstructured–Pytorch," https://pytorch.org/docs/stable/generated/torch.nn.utils.prune.L1Unstructured.html, accessed: 2021-05-14.

[45] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of bfloat16 for deep learning training," 2019.

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

[47] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017.

[48] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[49] "Quantization–Pytorch," https://pytorch.org/docs/stable/quantization.html, accessed: 2021-05-14.

[50] "Mixed-Precision Programming with CUDA 8," https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/, accessed: 2021-06-22.

[51] "Pytorch Deeplabv3+ implementation," https://github.com/VainF/DeepLabV3Plus-Pytorch, accessed: 2021-05-26.

[52] "ZCU104 Evaluation Board–User Guide," https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf, accessed: 2021-05-14.

[53] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, no. 2, p. 135–253, Feb. 2008. [Online]. Available: https://doi.org/10.1561/1000000005

[54] "Zynq UltraScale+ MPSoC Data Sheet:Overview," https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, accessed: 2021-05-14.

[55] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "Snap: An efficient sparse neural acceleration processor for unstructured sparse deep [-1pt] neural network inference," *IEEE Journal of Solid-State Circuits*, 2020.

List of Abbreviations

AI . . . . . Artificial Intelligence

ANN . . . . . Artificial neural network

CNN . . . . . Convolutional neural network

FPGA . . . . . Field programmable gate array

DRAM . . . . . Dynamic random-access memory

BRAM . . . . . Block random-access memory

ZeNA . . . . . Zero-aware neural network accelerator

VHDL . . . . . Very High Speed Integrated Circuit Hardware Description Language

PE . . . . . Processing Element

RF . . . . . Register file

PL . . . . . Programmable logic

PS . . . . . Processing Subsystem

psum . . . . . Partial sum

ifmap . . . . . Input feature map

ofmap . . . . . Output feature map

iact . . . . . Input activation

APPENDIX A

README

A.1 Running the accelerator

First the modified model of DeepLabv3 needs to be run in order to extract ifmap inputs, filters and ofmaps of the ASPP layer. This is done by executing the `deepLabv3main.py` script in the deeplabv3 directory.

The script is based on [1]. In order to run the script some setup is needed as described in [1].

Once the setup is completed for the Pascal VOC trainaug dataset the modified script can be run as follows:

```
$ python deepLabv3main.py -model deeplabv3_resnet101 -crop_val
-ckpt model/best_deeplabv3_resnet101_voc_os16.pth -year 2012_aug
-batch_size 16 -extract_values
```

This will run the model and will create some files in the local data folder. One of each file type (input, output, weights) needs to be placed in the `scripts` folder and needs to be renamed to `input_prunned.npy`, `outputs_prunned.npy` and `weights_prunned.npy`.

Next the `export_data.py` script can be executed. The calling interface looks as follows:

```
$ export_data.py [IFMAP_DEPTH/64] [PARALLEL_OFMS] [OFMS] [RATE] [REORDERED]
```

Importantly, the ifmap depth will be multiplied by 64. An example call with 640 ifmap channels, 32 parallel ofmap channels, 32 ofmap channels, an atrous rate of 6 and no re-ordering of the ofmap channels:

```
$ python export_data.py 10 32 32 6 false
```

Next, the provided source files need to be synthesis & implemented in Vivado 2021.2. For this it is necessary to create an UART and clock from the design libraries (both at 100Mhz). Once the bitstream is created it can be uploaded to the FPGA.

In order to catch the output of the FPGA over UART miniterm [2] can be utilized:

```
$ sudo miniterm -raw [port] | tee output.data
```

The resulting output file `output.data` needs to be post-processed by the `convert_to_result.py` script:

```
$ python convert_to_result.py output.data [IFMAP_DEPTH/64] [PARALLEL_OF
[OFMS] [PE_COLUMNS] [REORDERED]
```

This yields a `...-processed.data` file which contains the transformed outputs. This file should be equivalent to the results file yielded by running the `export_data.py` script.

[1] https://github.com/VainF/DeepLabV3Plus-Pytorch

[2] https://pyserial.readthedocs.io/en/latest/tools.html

APPENDIX B

Source Code

B.1 VHDL

core_pck.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  PACKAGE core_pck IS
6    CONSTANT FILTER_DEPTH : NATURAL := 1;
7    CONSTANT PARALLEL_OFMS : NATURAL := 1;
8    CONSTANT MAX_OFMS : NATURAL := 1;
9    CONSTANT PE_COLUMNS : NATURAL := 3;
10   CONSTANT OFM_REQUANT : NATURAL := 66;
11   CONSTANT IFMAP_ZERO_CONSTANT : NATURAL := 24;
12   CONSTANT DATA_WIDTH : NATURAL := 8;
13   CONSTANT FILTER_PER_PE : NATURAL := 64;
14   CONSTANT BUSSIZE : NATURAL := FILTER_PER_PE * DATA_WIDTH +
     ↪  FILTER_PER_PE; -- data +bitvec + ifmap zero offset + Ifmap_zero
15   CONSTANT MAX_RATE : NATURAL := 1;
16   CONSTANT ACC_DATA_WIDTH : NATURAL := 24;
17   CONSTANT EXEC_COUNTER_WIDTH : NATURAL := 32;
18
19   CONSTANT IFMAP_SIZE : NATURAL := 33;
20   CONSTANT DILATION_RATE : NATURAL := 6;
21
22 END PACKAGE;
```

project_top.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer: Fabian Kresse
4  --
5  -- Create Date: 03/17/2021 12:21:08 PM
6  -- Design Name:
7  -- Module Name: project_top - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 LIBRARY IEEE;
21 USE IEEE.STD_LOGIC_1164.ALL;
22 USE work.core_pck.ALL;
23
24 ENTITY project_top IS
25   PORT (
26     clk_in1_p : IN STD_LOGIC;
27     clk_in1_n : IN STD_LOGIC;
28     rx : IN STD_LOGIC;
29     tx : OUT STD_LOGIC
30   );
31 END project_top;
32
33 ARCHITECTURE Behavioral OF project_top IS
34   COMPONENT clk_wiz_0
35     PORT (
36       clk_out1 : OUT STD_LOGIC;
37       reset : IN STD_LOGIC;
38       locked : OUT STD_LOGIC;
39       clk_in1_p : IN STD_LOGIC;
40       clk_in1_n : IN STD_LOGIC
41     );
42   END COMPONENT;
43   SIGNAL clk, clk_out, reset, locked : STD_LOGIC;
44   CONSTANT reset_top : STD_LOGIC := '1';
45 BEGIN
46
47   CLOCK_100MHZ : clk_wiz_0
```

```
48    PORT MAP(
49      clk_out1 => clk_out,
50      reset => NOT(reset_top),
51      locked => locked,
52      clk_in1_p => clk_in1_p,
53      clk_in1_n => clk_in1_n
54    );
55
56    pro_top_i : ENTITY work.top
57      GENERIC MAP(
58        PARALLEL_OFMS => PARALLEL_OFMS,
59        MAX_OFMS => MAX_OFMS,
60        FILTER_DEPTH => FILTER_DEPTH,
61        FILTER_VALUES => FILTER_PER_PE,
62        MAX_RATE => MAX_RATE,
63        PE_COLUMNS => PE_COLUMNS,
64        OFM_REQUANT => OFM_REQUANT
65      )
66      PORT MAP(
67        reset => reset_top AND locked,
68        clk => clk_out,
69        rx => rx,
70        tx => tx
71      );
72  END Behavioral;
```

top.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   USE ieee.std_logic_misc.ALL;
5
6   USE work.core_pck.ALL;
7   USE work.top_types_pck.ALL;
8
9   ENTITY top IS
10    GENERIC (
11      PARALLEL_OFMS : NATURAL := 4; --variable
12      MAX_OFMS : NATURAL := 4; --variable
13      FILTER_DEPTH : NATURAL := 1; --32; --variable
14      FILTER_VALUES : NATURAL := 64; --fixed cannot be easily
        ↪ extended/changed
15      MAX_RATE : NATURAL := 1; --either 1,2,3 not yet extended beyond 1
16      PE_COLUMNS : NATURAL := 3; --fixed could be extended to 9
17      OFM_REQUANT : NATURAL := 62
18    );
19    PORT (
20      reset, clk : IN STD_LOGIC;
```

```
21      rx : IN STD_LOGIC;
22      tx : OUT STD_LOGIC
23    );
24  END ENTITY;
25
26  ARCHITECTURE arch OF top IS
27    --UART and OFMS_UNIT COMMUNICATION
28    SIGNAL from_uart : from_uart_type;
29    SIGNAL to_uart : to_uart_type;
30    SIGNAL to_uart_from_ofm, to_uart_from_counters : to_uart_type;
31    --ctrl to PEs
32    SIGNAL ctrl_to_PEs : ctrl_to_PEs_type;
33    --from bitvec generated
34    SIGNAL bus_to_array, bus_values : STD_LOGIC_VECTOR(BUSSIZE - 1
      ↪  DOWNTO 0);
35    SIGNAL new_kernels_to_ctrl : STD_LOGIC_VECTOR(PARALLEL_OFMS - 1
      ↪  DOWNTO 0);
36    --PEs to ctrl
37    SIGNAL PEs_finished : STD_LOGIC;
38    SIGNAL psums_from_array : psum_array;
39    SIGNAL array_finished_ifmaps : STD_LOGIC;
40    CONSTANT ifmap_zero_offset : STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO
      ↪  0) := STD_LOGIC_VECTOR(to_unsigned(IFMAP_ZERO_CONSTANT,
      ↪  DATA_WIDTH));
41    --used to calculate/output utilization
42    SIGNAL mult_counter : mult_counter_array;
43    SIGNAL finished_ofms_to_storage : ofms_out_type;
44    SIGNAL exc_counter : unsigned(EXEC_COUNTER_WIDTH - 1 DOWNTO 0);
45    SIGNAL uart_exec_data_counter, uart_exec_data_counter_nxt : NATURAL
      ↪  RANGE 0 TO PARALLEL_OFMS * PE_COLUMNS * 4 + 4 + 10;
46    SIGNAL uart_exec_data, uart_exec_data_nxt : STD_LOGIC_VECTOR(8 - 1
      ↪  DOWNTO 0);
47    SIGNAL finished_counters : STD_LOGIC;
48    SIGNAL slice_cp, slice_cp_nxt : NATURAL RANGE 0 TO 4 - 1;
49    SIGNAL ofm_cp, ofm_cp_nxt : NATURAL RANGE 0 TO PARALLEL_OFMS - 1;
50    SIGNAL column_cp, column_cp_nxt : NATURAL RANGE 0 TO PE_COLUMNS -
      ↪  1;
51    SIGNAL in_unit_to_ctrl : in_unit_to_ctrl_type;
52    SIGNAL ctrl_to_in : ctrl_to_in_type;
53    --everything finished
54    SIGNAL finished : STD_LOGIC;
55  BEGIN
56
57    -- The following processes are responsible for the UART control and
      ↪  outputing multiplication counters
58    -- (for benchmarking the utilization)
59    uart_c_sync : PROCESS (reset, clk)
60    BEGIN
61      IF reset = '0' THEN
```

```
62        uart_exec_data_counter <= 0;
63        uart_exec_data <= (OTHERS => '0');
64        slice_cp <= 0;
65        ofm_cp <= 0;
66        column_cp <= 0;
67      ELSIF rising_edge(clk) THEN
68        uart_exec_data_counter <= uart_exec_data_counter_nxt;
69        uart_exec_data <= uart_exec_data_nxt;
70        column_cp <= column_cp_nxt;
71        slice_cp <= slice_cp_nxt;
72        ofm_cp <= ofm_cp_nxt;
73      END IF;
74
75    END PROCESS;
76
77    uart_counter_gen : PROCESS (ALL)
78    BEGIN
79      to_uart_from_counters.valid <= '0';
80      to_uart_from_counters.data <= uart_exec_data;
81      uart_exec_data_counter_nxt <= uart_exec_data_counter;
82      IF from_uart.want_data_counters = '1' THEN
83        IF from_uart.ready = '1' THEN
84          to_uart_from_counters.valid <= '1';
85          to_uart_from_counters.data <= uart_exec_data;
86          uart_exec_data_counter_nxt <= uart_exec_data_counter + 1;
87        END IF;
88      END IF;
89    END PROCESS;
90
91    uart_exec_data_prov : PROCESS (ALL)
92    BEGIN
93      finished_counters <= '0';
94      uart_exec_data_nxt <= X"0A";
95      ofm_cp_nxt <= ofm_cp;
96      slice_cp_nxt <= slice_cp;
97      column_cp_nxt <= column_cp;
98      IF uart_exec_data_counter < 4 THEN
99        uart_exec_data_nxt <= STD_LOGIC_VECTOR(exc_counter(DATA_WIDTH *
          ↪  (4 - (uart_exec_data_counter)) - 1 DOWNTO DATA_WIDTH * (3 -
          ↪  (uart_exec_data_counter))));
100     ELSIF uart_exec_data_counter < PARALLEL_OFMS * PE_COLUMNS * 4 + 4
        ↪  THEN
101       IF from_uart.ready = '1' THEN
102         IF slice_cp = 4 - 1 THEN
103           slice_cp_nxt <= 0;
104           IF column_cp = PE_COLUMNS - 1 THEN
105             column_cp_nxt <= 0;
106             IF ofm_cp = PARALLEL_OFMS - 1 THEN
107               ofm_cp_nxt <= 0;
```

```
108               ELSE
109                 ofm_cp_nxt <= ofm_cp + 1;
110               END IF;
111             ELSE
112               column_cp_nxt <= column_cp + 1;
113             END IF;
114           ELSE
115             slice_cp_nxt <= slice_cp + 1;
116           END IF;
117         END IF;
118         uart_exec_data_nxt <= STD_LOGIC_VECTOR(mult_counter(column_cp,
            ↪  ofm_cp)(DATA_WIDTH * (4 - (slice_cp)) - 1 DOWNTO DATA_WIDTH
            ↪  * (3 - (slice_cp))));
119       ELSIF uart_exec_data_counter = PARALLEL_OFMS * PE_COLUMNS * 4 + 4
          ↪  THEN
120         uart_exec_data_nxt <= X"0A";
121       ELSE
122         finished_counters <= '1';
123       END IF;
124     END PROCESS;
125     uart_arb : PROCESS (ALL)
126     BEGIN
127       IF from_uart.want_data_ofm = '1' THEN
128         to_uart <= to_uart_from_ofm;
129       ELSE
130         to_uart <= to_uart_from_counters;
131       END IF;
132     END PROCESS;
133
134     -- The output feature map unit stores the completed psums and
        ↪  requantizes them
135     requant_unit : ENTITY work.ofms_unit
136       GENERIC MAP(
137         PARALLEL_OFMS => PARALLEL_OFMS,
138         MAX_OFMS => MAX_OFMS,
139         MAX_RATE => MAX_RATE,
140         PE_COLUMNS => PE_COLUMNS,
141         OFM_REQUANT => OFM_REQUANT
142       )
143       PORT MAP(
144         clk => clk,
145         reset => reset,
146         ofms_in => finished_ofms_to_storage,
147         from_uart => from_uart,
148         to_uart => to_uart_from_ofm
149       );
150
151     -- The uart_unit is the master in the comm with the ofm_unit
```

```
152     -- it tells the ofm_unit once data should be prepared and when to
        ↪  send
153     -- new data
154     uart_i : ENTITY work.uart_unit
155       PORT MAP(
156         clk => clk,
157         reset => reset,
158         from_uart => from_uart,
159         to_uart => to_uart,
160         rx => rx,
161         tx => tx,
162         finished => finished, --just used for asserting
            ↪  from_uart.want_data <= '1'
163         finished_counters => finished_counters
164       );
165
166     -- stores ifmaps and kernels at startup
167     in_unit_i : ENTITY work.in_unit
168       GENERIC MAP(
169         PARALLEL_OFMS => PARALLEL_OFMS,
170         MAX_OFMS => MAX_OFMS,
171         MAX_RATE => MAX_RATE,
172         PE_COLUMNS => PE_COLUMNS,
173         FILTER_DEPTH => FILTER_DEPTH
174       )
175       PORT MAP(
176         clk => clk,
177         reset => reset,
178         in_unit_to_ctrl => in_unit_to_ctrl,
179         ctrl_to_in => ctrl_to_in
180       );
181
182     --responsible for the control flow
183     cntrl_unit_i : ENTITY work.cntrl_unit
184       GENERIC MAP(
185         PARALLEL_OFMS => PARALLEL_OFMS,
186         MAX_OFMS => MAX_OFMS,
187         FILTER_DEPTH => FILTER_DEPTH,
188         FILTER_VALUES => FILTER_VALUES,
189         MAX_RATE => MAX_RATE
190       )
191       PORT MAP(
192         clk => clk,
193         reset => reset,
194         ctrl_to_in => ctrl_to_in,
195         in_unit_to_ctrl => in_unit_to_ctrl,
196         ctr_to_PEs => ctrl_to_PEs,
197         PEs_finished => array_finished_ifmaps,
198         psum_values_in => psums_from_array,
```

```
199         ofms_out => finished_ofms_to_storage,
200         finished => finished,
201         exc_counter => exc_counter
202       );
203     --creates the zero/non-zero bitvectors
204     bitvec_i : ENTITY work.bitvec
205       GENERIC MAP(
206         PARALLEL_OFMS => PARALLEL_OFMS,
207         FILTER_VALUES => FILTER_VALUES,
208         PE_COLUMNS => PE_COLUMNS
209       )
210       PORT MAP(
211         clk => clk,
212         reset => reset,
213         kernels_to_bitvec => ctrl_to_PEs.kernel_values,
214         iacts_to_bitvec => ctrl_to_PEs.ifmap_values,
215         new_ifmaps => ctrl_to_PEs.new_ifmaps,
216         new_kernels => ctrl_to_PEs.new_kernels,
217         bus_values => bus_values,
218         ifmap_zero_offset => ifmap_zero_offset
219       );
220
221     -- The PE array
222     pe_array_i : ENTITY work.pe_array
223       GENERIC MAP(
224         PARALLEL_OFMS => PARALLEL_OFMS,
225         FILTER_DEPTH => FILTER_DEPTH,
226         PE_COLUMNS => PE_COLUMNS -- fixed, could be extended
227       )
228       PORT MAP(
229         reset => reset,
230         clk => clk,
231         bus_pe_array => bus_to_array,
232         new_kernels_to_array => ctrl_to_PEs.new_kernels,
233         new_ifmaps_to_array => ctrl_to_PEs.new_ifmaps,
234         psums_to_control => psums_from_array,
235         psums_from_control => ctrl_to_PEs.new_psum_values,
236         get_psums => ctrl_to_PEs.get_psums,
237         new_psum => ctrl_to_PEs.new_psums,
238         ifmap_zero_offset => ifmap_zero_offset,
239         finished_ifmaps_out => array_finished_ifmaps,
240         mult_counter => mult_counter
241       );
242     bus_arb : PROCESS (ALL)
243     BEGIN
244
245       IF OR_REDUCE(ctrl_to_PEs.new_ifmaps) = '1' OR
        ↪  OR_REDUCE(ctrl_to_PEs.new_kernels) = '1' THEN
246         bus_to_array <= bus_values;
```

```
247       ELSE
248         bus_to_array <= (OTHERS => '-');
249       END IF;
250
251     END PROCESS;
252   END ARCHITECTURE;
```

top_types_pck.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4
5   USE work.core_pck.ALL;
6
7   PACKAGE top_types_pck IS
8     TYPE psum_array IS ARRAY(0 TO PARALLEL_OFMS - 1, 0 TO PE_COLUMNS -
      ↪  1) OF signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
9     TYPE iact_values_array IS ARRAY(0 TO FILTER_PER_PE - 1) OF
      ↪  unsigned(DATA_WIDTH - 1 DOWNTO 0);
10    TYPE kernel_values_array IS ARRAY(0 TO FILTER_PER_PE - 1) OF
      ↪  signed(DATA_WIDTH - 1 DOWNTO 0);
11
12    TYPE ifmap_DRAM_type IS RECORD
13      valid : STD_LOGIC;
14      data : STD_LOGIC_VECTOR(72 * 8 - 1 DOWNTO 0);
15    END RECORD;
16
17    TYPE ofms_out_type IS RECORD
18      data : STD_LOGIC_VECTOR(PARALLEL_OFMS * PE_COLUMNS *
        ↪  ACC_DATA_WIDTH - 1 DOWNTO 0);
19      valid : STD_LOGIC;
20    END RECORD;
21    TYPE from_uart_type IS RECORD
22      want_data_ofm : STD_LOGIC;
23      want_data_counters : STD_LOGIC;
24      ready : STD_LOGIC;
25    END RECORD;
26
27    TYPE to_uart_type IS RECORD
28      data : STD_LOGIC_VECTOR(7 DOWNTO 0);
29      valid : STD_LOGIC;
30    END RECORD;
31
32    --just used for determining the utilization
33    TYPE mult_counter_array IS ARRAY(0 TO PE_COLUMNS - 1, 0 TO
      ↪  PARALLEL_OFMS - 1) OF unsigned(EXEC_COUNTER_WIDTH - 1 DOWNTO
      ↪  0);
34
```

```
35    TYPE ctrl_to_PEs_type IS RECORD
36      new_ifmaps : STD_LOGIC_VECTOR(PE_COLUMNS - 1 DOWNTO 0);
37      new_kernels : STD_LOGIC_VECTOR(PARALLEL_OFMS - 1 DOWNTO 0);
38      get_psums : STD_LOGIC;
39      new_psums : STD_LOGIC;
40      new_psum_values : psum_array;
41      kernel_values : kernel_values_array;
42      ifmap_values : iact_values_array;
43    END RECORD;
44
45    TYPE in_unit_to_ctrl_type IS RECORD
46      ifmap_values : ifmap_DRAM_type; -- ifmap_values
47      ifmaps_loaded : STD_LOGIC; -- all ifmaps have been loaded and are
        ↪ completly written to the ifmap mem
48      kernel_values : kernel_values_array; -- values of the kernels
49      kernels_loaded : STD_LOGIC; -- all kernels of current position
        ↪ have been provided
50      new_kernels : STD_LOGIC_VECTOR(PARALLEL_OFMS - 1 DOWNTO 0); --
        ↪ what kernels was loaded
51    END RECORD;
52
53    TYPE ctrl_to_in_type IS RECORD
54      load_ifmaps : STD_LOGIC; -- load ifmaps next
55      load_kernels : STD_LOGIC; -- load kernels next
56    END RECORD;
57
58
59  END PACKAGE;
```

pe_array.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   USE ieee.std_logic_misc.ALL;
5   USE work.core_pck.ALL;
6   USE work.top_types_pck.ALL;
7   USE work.pe_array_pck.ALL;
8
9   ENTITY pe_array IS
10    GENERIC (
11      PARALLEL_OFMS : NATURAL := 3;
12      FILTER_DEPTH : NATURAL := 32;
13      PE_COLUMNS : NATURAL := 3
14    );
15    PORT (
16      reset, clk : IN STD_LOGIC;
17      bus_pe_array : IN STD_LOGIC_VECTOR(BUSSIZE - 1 DOWNTO 0);
```

```
18      new_kernels_to_array : IN STD_LOGIC_VECTOR(PARALLEL_OFMS - 1
        ↪  DOWNTO 0);
19      new_ifmaps_to_array : IN STD_LOGIC_VECTOR(PE_COLUMNS - 1 DOWNTO
        ↪  0);
20      psums_to_control : OUT psum_array;
21      psums_from_control : IN psum_array;
22      get_psums : IN STD_LOGIC;
23      new_psum : IN STD_LOGIC;
24      ifmap_zero_offset : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
25      finished_ifmaps_out : OUT STD_LOGIC;
26      mult_counter : OUT mult_counter_array
27    );
28  END ENTITY;
29
30  ARCHITECTURE arch OF pe_array IS
31
32    SIGNAL finished_ifmaps, new_kernels, new_ifmaps : std_logic_array;
33    TYPE psums_array IS ARRAY(0 TO PE_COLUMNS - 1, 0 TO PARALLEL_OFMS -
      ↪  1) OF signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
34
35    SIGNAL psum : psums_array;
36    SIGNAL psums_bus : STD_LOGIC_VECTOR(BUSSIZE - 1 DOWNTO 0);
37    SIGNAL bus_to_pe : STD_LOGIC_VECTOR(BUSSIZE - 1 DOWNTO 0);
38    SIGNAL psum_in : psums_array;
39
40  BEGIN
41    -- 9 = PE_COLUMNS could be added somewhat easily, other values are
      ↪  more problematic
42    --ASSERT PE_COLUMNS = 3 or PE_COLUMNS = 2 or PE_COLUMNS = 1 REPORT
      ↪  "Only 1, 2 or 3 PE_COLUMNS are supported!";
43    --ASSERT IFMAP_SIZE mod PE_COLUMNS = 0 REPORT "IFMAP SIZE MUST BE
      ↪  DIVISIBLE by PE_COLUMNS!";
44    PEs_rows : FOR row IN 0 TO PARALLEL_OFMS - 1 GENERATE
45      PEs_columns : FOR col IN 0 TO PE_COLUMNS - 1 GENERATE
46        pe_i : ENTITY work.pe
47          PORT MAP(
48            reset => reset,
49            clk => clk,
50            new_kernels => new_kernels(col, row),
51            new_ifmaps => new_ifmaps(col, row),
52            new_psum => new_psum,
53            psum_in => psum_in(col, row),
54            bus_to_pe => bus_to_pe,
55            psum => psum(col, row),
56            mult_counter => mult_counter(col, row),
57            ifmap_zero_offset => ifmap_zero_offset,
58            finished_ifmaps => finished_ifmaps(col, row)
59          );
60      END GENERATE;
```

```
61    END GENERATE;
62
63    -- output all PEs finished
64    out_p : PROCESS (ALL)
65    BEGIN
66      finished_ifmaps_out <= AND_REDUCE_COL_ROWS(finished_ifmaps);
67    END PROCESS;
68
69    -- Selects the appropiate PEs for receiving new values
70    bus_driver : PROCESS (ALL)
71    BEGIN
72      new_kernels <= (OTHERS => (OTHERS => '0'));
73      new_ifmaps <= (OTHERS => (OTHERS => '0'));
74      bus_to_pe <= bus_pe_array WHEN OR_REDUCE(new_ifmaps_to_array) =
        ↪  '1' OR OR_REDUCE(new_kernels_to_array) = '1' OR new_psum =
        ↪  '1' ELSE
75        (OTHERS => '-');
76
77      FOR row IN 0 TO PARALLEL_OFMS - 1 LOOP
78        IF new_kernels_to_array(row) = '1' THEN
79          FOR col IN 0 TO PE_COLUMNS - 1 LOOP
80            new_kernels(col, row) <= '1';
81          END LOOP;
82        END IF;
83      END LOOP;
84
85      FOR col IN 0 TO PE_COLUMNS - 1 LOOP
86        IF new_ifmaps_to_array(col) = '1' THEN
87          FOR row IN 0 TO PARALLEL_OFMS - 1 LOOP
88            new_ifmaps(col, row) <= '1';
89          END LOOP;
90        END IF;
91      END LOOP;
92    END PROCESS;
93
94    -- sends the psums to the right PEs
95    psum_input : PROCESS (ALL)
96    BEGIN
97      psum_in <= (OTHERS => (OTHERS => (OTHERS => '0')));
98      FOR row IN 0 TO PARALLEL_OFMS - 1 LOOP
99        FOR col IN 0 TO PE_COLUMNS - 1 LOOP
100         psum_in(col, row) <= psums_from_control(row, col);
101       END LOOP;
102     END LOOP;
103   END PROCESS;
104
105   --writes the psums back to the psums stage
106   psums_back : PROCESS (ALL)
107   BEGIN
```

```
108       psums_bus <= (OTHERS => '0');
109       FOR row IN 0 TO PARALLEL_OFMS - 1 LOOP
110         FOR col IN 0 TO PE_COLUMNS - 1 LOOP
111           psums_to_control(row, col) <= psum(col, row);
112         END LOOP;
113       END LOOP;
114     END PROCESS;
115   END ARCHITECTURE;
```

pe_array_pck.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   USE work.core_pck.ALL;
5   USE work.pe_pack.ALL;
6
7   PACKAGE pe_array_pck IS
8     TYPE std_logic_array IS ARRAY(0 TO PE_COLUMNS - 1, 0 TO
      ↪  PARALLEL_OFMS - 1) OF STD_LOGIC;
9     FUNCTION AND_REDUCE_COL_ROWS(v : std_logic_array) RETURN STD_LOGIC;
10    FUNCTION OR_REDUCE_COL_ROWS(v : std_logic_array) RETURN STD_LOGIC;
11
12  END PACKAGE;
13
14  PACKAGE BODY pe_array_pck IS
15
16    FUNCTION AND_REDUCE_COL_ROWS(v : std_logic_array) RETURN STD_LOGIC
      ↪  IS
17    BEGIN
18      FOR I IN 0 TO PE_COLUMNS - 1 LOOP
19        FOR J IN 0 TO PARALLEL_OFMS - 1 LOOP
20          IF v(I, J) = '0' THEN
21            RETURN '0';
22          END IF;
23        END LOOP;
24      END LOOP;
25      RETURN '1';
26    END FUNCTION;
27
28    FUNCTION OR_REDUCE_COL_ROWS(v : std_logic_array) RETURN STD_LOGIC
      ↪  IS
29    BEGIN
30      FOR I IN 0 TO PE_COLUMNS - 1 LOOP
31        FOR J IN 0 TO PARALLEL_OFMS - 1 LOOP
32          IF v(I, J) = '1' THEN
33            RETURN '1';
34          END IF;
35        END LOOP;
```

```
36      END LOOP;
37      RETURN '0';
38    END FUNCTION;
39
40  END PACKAGE BODY;
```

pe_pck.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   USE work.core_pck.ALL;
5
6   PACKAGE pe_pack IS
7     CONSTANT COMPARISON_BITVEC_WIDTH : NATURAL := 16;
8     CONSTANT DATA_WIDTH_RESULT : NATURAL := 18;
9   END PACKAGE;
```

bitvec_unit.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   USE ieee.std_logic_misc.ALL;
5   USE work.core_pck.ALL;
6   USE work.control_pck.ALL;
7   USE work.top_types_pck.ALL;
8   USE work.pe_array_pck.ALL;
9
10  ENTITY bitvec IS
11    GENERIC (
12      PARALLEL_OFMS : NATURAL := 4;
13      FILTER_VALUES : NATURAL := 64;
14      PE_COLUMNS : NATURAL := 3
15    );
16    PORT (
17      clk, reset : IN STD_LOGIC;
18      kernels_to_bitvec : IN kernel_values_array;
19      iacts_to_bitvec : IN iact_values_array;
20      new_ifmaps : IN STD_LOGIC_VECTOR(PE_COLUMNS - 1 DOWNTO 0);
21      new_kernels : IN STD_LOGIC_VECTOR(PARALLEL_OFMS - 1 DOWNTO 0);
22      bus_values : OUT STD_LOGIC_VECTOR(BUSSIZE - 1 DOWNTO 0);
23      ifmap_zero_offset : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0)
24    );
25  END ENTITY;
26
27  --returns the bus values and encodes zeros in the bitvev arrays
28  ARCHITECTURE arch OF bitvec IS
29    SIGNAL zero_point : NATURAL RANGE 0 TO 255 - 1;
```

```
30 BEGIN
31   fow : PROCESS (ALL)
32   BEGIN
33     bus_values <= (OTHERS => '0');
34 
35     IF OR_REDUCE (new_ifmaps) = '1' THEN
36       FOR I IN 0 TO 63 LOOP
37         IF to_integer(unsigned(iacts_to_bitvec(I))) = zero_point THEN
38           bus_values(I) <= '0';
39         ELSE
40           bus_values(I) <= '1';
41         END IF;
42         bus_values(DATA_WIDTH * (I + 1) + 63 DOWNTO DATA_WIDTH * I +
           ↪  64) <= STD_LOGIC_VECTOR(iacts_to_bitvec(I));
43       END LOOP;
44     ELSIF OR_REDUCE(new_kernels) = '1' THEN
45       FOR I IN 0 TO 63 LOOP
46         IF to_integer(signed(kernels_to_bitvec(I))) = 0 THEN
47           bus_values(I) <= '0';
48         ELSE
49           bus_values(I) <= '1';
50         END IF;
51         bus_values(DATA_WIDTH * (I + 1) + 63 DOWNTO DATA_WIDTH * I +
           ↪  64) <= STD_LOGIC_VECTOR(kernels_to_bitvec(I));
52       END LOOP;
53     END IF;
54   END PROCESS;
55 
56   zero_offs_sync : PROCESS (clk, reset)
57   BEGIN
58     IF reset = '0' THEN
59       zero_point <= 0;
60     ELSIF rising_edge(clk) THEN
61       zero_point <= to_integer(unsigned(ifmap_zero_offset));
62     END IF;
63   END PROCESS;
64 
65 END ARCHITECTURE;
```

control/ctrl.vhd

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE ieee.std_logic_misc.ALL;
5 
6 USE work.core_pck.ALL;
7 USE work.control_pck.ALL;
8 USE work.top_types_pck.ALL;
```

```
9
10 USE IEEE.math_real.ALL;
11
12 ENTITY cntrl_unit IS
13   GENERIC (
14     PARALLEL_OFMS : NATURAL := 3;
15     MAX_OFMS : NATURAL := 255;
16     FILTER_DEPTH : NATURAL := 32;
17     FILTER_VALUES : NATURAL := 64;
18     MAX_RATE : NATURAL := 3
19   );
20   PORT (
21     clk, reset : IN STD_LOGIC;
22     --one to DRAM/kernel unit
23     ctrl_to_in : OUT ctrl_to_in_type;
24     in_unit_to_ctrl : IN in_unit_to_ctrl_type;
25     --load_ifmaps : OUT STD_LOGIC;
26     --load_kernels : OUT STD_LOGIC;
27     --from DRAM
28
29     --kernels_loaded : IN STD_LOGIC;
30     --ifmaps_loaded : IN STD_LOGIC;
31     --ifmap_values_from_dram : IN ifmap_DRAM_type;
32     --kernel_values_from_dram : IN kernel_values_array;
33     --control to PEs
34     --new_kernels_valid : IN STD_LOGIC_VECTOR(PARALLEL_OFMS - 1
       ↪  DOWNTO 0);
35     ctr_to_PEs : OUT ctrl_to_PEs_type;
36     --control from PEs
37     PEs_finished : IN STD_LOGIC;
38     --values from PE array
39     psum_values_in : IN psum_array;
40     ofms_out : OUT ofms_out_type;
41     --values to PE arry
42     exc_counter : OUT unsigned(EXEC_COUNTER_WIDTH - 1 DOWNTO 0);
43     finished : OUT STD_LOGIC
44   );
45 END ENTITY;
46
47 ARCHITECTURE arch OF cntrl_unit IS
48
49   --control state
50   TYPE state_type IS (LOADING_IFMAPS, WAITING, WRITE_IFMAP,
     ↪  WRITE_KERNEL, CLEAN, WRITE_PSUMS_OUT, FINISHED_STATE);
51   SIGNAL state, state_nxt : state_type;
52
53   --control for input activation buffer
54   SIGNAL iacts_buffer_mode : iacts_mode_type;
55   SIGNAL ifmap_out_buffer : iacts_buffer_type;
```

```
56    --ifmap out buffer valid if high
57    SIGNAL ifmaps_prepared : STD_LOGIC;
58    --counts the number of ifmaps writen
59    SIGNAL write_ifmap_counter, write_ifmap_counter_nxt : NATURAL RANGE
      ↪  0 TO PE_COLUMNS - 1;
60
61    --signals the the in unit should provide a kernel next
62    SIGNAL need_kernel_nxt, need_kernel : STD_LOGIC;
63    SIGNAL need_new_kernel : STD_LOGIC;
64    --the current position in the ifmap/kernel
65    SIGNAL ifmap_position, ifmap_position_prev : ifmap_position_type;
66    SIGNAL write_kernel_counter, write_kernel_counter_nxt : NATURAL
      ↪  RANGE 0 TO PARALLEL_OFMS;
67    --controls the psum buffer/its outputs
68    TYPE psums_state_type IS (REQUEST_PSUMS, FETCH, PROCESS_PSUMS,
      ↪  IDLE);
69    SIGNAL psum_state_nxt, psum_state : psums_state_type;
70    SIGNAL psum_mode : mode_psums_type;
71    SIGNAL psums_position : point;
72    SIGNAL psums_position_prev : point;
73    SIGNAL psums_writen, psums_writen_nxt : STD_LOGIC;
74    -- high on buffer out valid
75    SIGNAL psums_ready : STD_LOGIC;
76    SIGNAL psum_values_out : psum_array;
77    --singals that psums buffer should only provide 0s
78    SIGNAL first_pass : STD_LOGIC;
79
80    SIGNAL write_out_ofms, finished_writing_ofm : STD_LOGIC;
81    SIGNAL finished_all : STD_LOGIC;
82
83  BEGIN
84
85    sync : PROCESS (clk, reset)
86    BEGIN
87      IF reset = '0' THEN
88        state <= LOADING_IFMAPS;
89        need_kernel <= '1';
90        write_ifmap_counter <= 0;
91        psum_state <= REQUEST_PSUMS;
92        write_kernel_counter <= 0;
93        psums_writen <= '0';
94      ELSIF rising_edge(clk) THEN
95        state <= state_nxt;
96        need_kernel <= need_kernel_nxt;
97        write_ifmap_counter <= write_ifmap_counter_nxt;
98        psum_state <= psum_state_nxt;
99        write_kernel_counter <= write_kernel_counter_nxt;
100       psums_writen <= psums_writen_nxt;
101     END IF;
```

```
102   END PROCESS;
103
104   -- only needed for utilization measurement
105   exec_count : PROCESS (ALL)
106   BEGIN
107     IF reset = '0' THEN
108       exc_counter <= (OTHERS => '0');
109     ELSIF rising_edge(clk) THEN
110       exc_counter <= exc_counter;
111       IF finished_all = '0' THEN
112         IF NOT(state = LOADING_IFMAPS) THEN
113           exc_counter <= exc_counter + 1;
114         END IF;
115       END IF;
116     END IF;
117   END PROCESS;
118
119   --computes the state
120   state_pro : PROCESS (ALL)
121   BEGIN
122     state_nxt <= state;
123     need_kernel_nxt <= need_kernel;
124     ctrl_to_in.load_kernels <= '0';
125     write_ifmap_counter_nxt <= write_ifmap_counter;
126     write_kernel_counter_nxt <= write_kernel_counter;
127     ctrl_to_in.load_ifmaps <= '0';
128     finished <= '0';
129     CASE(state) IS
130
131       WHEN LOADING_IFMAPS =>
132       ctrl_to_in.load_ifmaps <= '1';
133       IF in_unit_to_ctrl.ifmaps_loaded = '1' THEN
134         state_nxt <= WAITING;
135       END IF;
136
137       --PROCESSING
138       WHEN WAITING =>
139       --ready for new values?
140       IF PEs_finished = '1' THEN
141         IF need_kernel = '1' AND psums_writen = '1' THEN
142           state_nxt <= WRITE_KERNEL;
143         ELSE
144           IF ifmaps_prepared = '1' AND psums_writen = '1' THEN
145             state_nxt <= WRITE_IFMAP;
146           END IF;
147         END IF;
148         IF write_out_ofms = '1' AND finished_writing_ofm = '0' AND
          ↪  psums_writen = '1' THEN
149           state_nxt <= WRITE_PSUMS_OUT;
```

```
150           END IF;
151         END IF;
152
153         --PROCESSING
154         WHEN WRITE_KERNEL =>
155         ctrl_to_in.load_kernels <= '1';
156         IF in_unit_to_ctrl.kernels_loaded = '1' AND
            ↪  write_kernel_counter >= PARALLEL_OFMS - 1 THEN
157           IF ifmaps_prepared = '1' THEN
158             state_nxt <= WRITE_IFMAP;
159             write_kernel_counter_nxt <= write_kernel_counter;
160           ELSE
161             write_kernel_counter_nxt <= write_kernel_counter;
162             ctrl_to_in.load_kernels <= '0';
163           END IF;
164         ELSE
165           IF write_kernel_counter >= PARALLEL_OFMS - 1 THEN
166             write_kernel_counter_nxt <= 0;
167           ELSE
168             write_kernel_counter_nxt <= write_kernel_counter + 1;
169           END IF;
170         END IF;
171
172         --PROCESSING
173         WHEN WRITE_PSUMS_OUT =>
174         IF finished_writing_ofm = '1' THEN
175           IF finished_all = '0' THEN
176             IF ifmaps_prepared = '1' THEN
177               state_nxt <= WRITE_IFMAP;
178             END IF;
179           ELSE
180             state_nxt <= FINISHED_STATE;
181           END IF;
182         END IF;
183
184         --PROCESSING
185         WHEN WRITE_IFMAP =>
186         write_kernel_counter_nxt <= 0;
187         IF write_ifmap_counter = PE_COLUMNS - 1 THEN
188           write_ifmap_counter_nxt <= 0;
189           state_nxt <= CLEAN;
190         ELSE
191           write_ifmap_counter_nxt <= write_ifmap_counter + 1;
192         END IF;
193
194         --PROCESSING
195         WHEN CLEAN =>
196         need_kernel_nxt <= need_new_kernel;
197         state_nxt <= WAITING;
```

```
198
199         WHEN FINISHED_STATE =>
200         state_nxt <= FINISHED_STATE;
201         finished <= '1';
202
203       END CASE;
204
205     END PROCESS;
206
207     --takes care of writing to the bitvec unit
208     write_to_pe_array : PROCESS (ALL)
209     BEGIN
210       ctr_to_PEs.ifmap_values <= (OTHERS => (OTHERS => '-'));
211       ctr_to_PEs.kernel_values <= (OTHERS => (OTHERS => '-'));
212       ctr_to_PEs.new_kernels <= (OTHERS => '0');
213
214       CASE(state) IS
215
216         WHEN WAITING =>
217
218         WHEN WRITE_IFMAP =>
219         FOR I IN 0 TO FILTER_PER_PE - 1 LOOP
220           ctr_to_PEs.ifmap_values(I) <=
              ↪  ifmap_out_buffer(write_ifmap_counter, I);
221         END LOOP;
222         WHEN WRITE_KERNEL =>
223         ctr_to_PEs.kernel_values <= in_unit_to_ctrl.kernel_values;
224         ctr_to_PEs.new_kernels <= in_unit_to_ctrl.new_kernels;
225         WHEN OTHERS =>
226
227       END CASE;
228
229     END PROCESS;
230
231     --this process controls the ifmap loading with the iacts_buffer
232     ifmaps_process : PROCESS (state, write_ifmap_counter) --all doesnt
        ↪  work here due to vivado bug
233     BEGIN
234       iacts_buffer_mode <= PREPARE_IFMAP;
235       ctr_to_PEs.new_ifmaps <= (OTHERS => '0');
236       CASE(state) IS
237         WHEN LOADING_IFMAPS =>
238         iacts_buffer_mode <= LOAD_IFMAP;
239         WHEN WAITING =>
240         iacts_buffer_mode <= PREPARE_IFMAP;
241         WHEN WRITE_KERNEL =>
242         iacts_buffer_mode <= PREPARE_IFMAP;
243         WHEN WRITE_IFMAP =>
244         ctr_to_PEs.new_ifmaps(write_ifmap_counter) <= '1';
```

```
245         WHEN CLEAN =>
246         iacts_buffer_mode <= CLEAN;
247         WHEN OTHERS =>
248       END CASE;
249     END PROCESS;
250 
251     --this process controls the psums buffer
252     psums_ctrl : PROCESS (ALL)
253     BEGIN
254       psum_mode <= CLEAN;
255       psum_state_nxt <= psum_state;
256       psums_writen_nxt <= psums_writen;
257       ctr_to_PEs.new_psums <= '0';
258       ctr_to_PEs.get_psums <= '0';
259       ctr_to_PEs.new_psum_values <= (OTHERS => (OTHERS => (OTHERS =>
          ↪  '-')));
260       CASE(state) IS
261         WHEN WRITE_PSUMS_OUT =>
262         psum_mode <= WRITE_OUT_PSUMS;
263         WHEN WAITING =>
264         CASE (psum_state) IS
265           WHEN REQUEST_PSUMS =>
266             ctr_to_PEs.get_psums <= '1';
267             psum_state_nxt <= FETCH;
268             psum_mode <= FETCH_PSUMS;
269           WHEN FETCH =>
270             psum_state_nxt <= PROCESS_PSUMS;
271             psum_mode <= FETCH_PSUMS;
272           WHEN PROCESS_PSUMS =>
273             psum_mode <= PREPARE_PSUMS;
274             psum_state_nxt <= PROCESS_PSUMS;
275             IF psums_ready = '1' THEN
276               ctr_to_PEs.new_psum_values <= psum_values_out;
277               psums_writen_nxt <= '1';
278               ctr_to_PEs.new_psums <= '1';
279               psum_state_nxt <= IDLE;
280             END IF;
281           WHEN OTHERS =>
282 
283         END CASE;
284         WHEN OTHERS =>
285         psum_state_nxt <= REQUEST_PSUMS;
286         psums_writen_nxt <= '0';
287       END CASE;
288     END PROCESS;
289 
290     --unit calculates the needed ifmap/psum positions
291     position_unit : ENTITY work.state_calc
292       GENERIC MAP(
```

```
293         PARALLEL_OFMS => PARALLEL_OFMS,
294         MAX_OFMS => MAX_OFMS,
295         FILTER_DEPTH => FILTER_DEPTH,
296         FILTER_VALUES => FILTER_VALUES,
297         MAX_RATE => MAX_RATE
298       )
299       PORT MAP(
300         clk => clk,
301         reset => reset,
302         new_ifmaps => OR_REDUCE(ctr_to_PEs.new_ifmaps), --new_ifmap
          ↪  event signals advance for state
303         need_kernel => need_new_kernel, --goes high if new kernel
          ↪  should be provided
304         first_pass => first_pass, --high on first pass tells
          ↪  psums_buffer to output 0s
305         write_out_ofms_out => write_out_ofms, --high starts writing out
          ↪  to storage
306         ifmap_position => ifmap_position, --the next ifmap_position
307         psums_position => psums_position,
308         psums_position_prev => psums_position_prev,
309         finished => finished_all
310       );
311
312     iacts_buffer_i : ENTITY work.iacts_buffer
313       GENERIC MAP(
314         IFMAP_SIZE => IFMAP_SIZE,
315         IFMAPS_TO_PREPARE => PE_COLUMNS,
316         AWIDTH => AWIDTH_IACTS_BUFFER,
317         DEPTH => DEPTH_IACTS_BUFFER
318       )
319       PORT MAP(
320         clk => clk,
321         reset => reset,
322         mode => iacts_buffer_mode,
323         values => in_unit_to_ctrl.ifmap_values,
324         address => ifmap_position,
325         out_buffer => ifmap_out_buffer,
326         ifmaps_prepared => ifmaps_prepared
327       );
328
329     psums_buffer_i : ENTITY work.psums_buffer
330       GENERIC MAP(
331         ACC_WIDTH => ACC_DATA_WIDTH,
332         MAX_OFMS => MAX_OFMS,
333         PE_COLUMNS => PE_COLUMNS,
334         PARALLEL_OFMS => PARALLEL_OFMS,
335         A_WIDTH => AWIDTH_PSUM_BUFFER,
336         DWIDTH => DWIDTH_PSUM_BUFFER,
337         DEPTH => DEPTH_PSUM_BUFFER
```

```
338     )
339     PORT MAP(
340       clk => clk,
341       reset => reset,
342       mode => psum_mode,
343       address => psums_position,
344       address_prev => psums_position_prev,
345       psums_ready => psums_ready,
346       psum_values_in => psum_values_in,
347       first_pass => first_pass,
348       out_buffer => psum_values_out,
349       ofms_out => ofms_out,
350       finished_writing_ofm => finished_writing_ofm
351     );
352 END ARCHITECTURE;
```

control/control_pck.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  USE work.core_pck.ALL;
6  USE IEEE.math_real.ALL;
7
8  PACKAGE control_pck IS
9
10   CONSTANT DEPTH_PSUM_BUFFER : NATURAL := IFMAP_SIZE *
     ↪  (IFMAP_SIZE/PE_COLUMNS); --33*11
11   CONSTANT AWIDTH_PSUM_BUFFER : NATURAL :=
     ↪  INTEGER(ceil(log2(real(DEPTH_PSUM_BUFFER))));
12   CONSTANT DWIDTH_PSUM_BUFFER : NATURAL := PARALLEL_OFMS *
     ↪  ACC_DATA_WIDTH * PE_COLUMNS;
13
14
15   CONSTANT AWIDTH_IACTS_BUFFER: NATURAL :=
     ↪  INTEGER(ceil(log2(real(FILTER_DEPTH * IFMAP_SIZE * IFMAP_SIZE *
     ↪  2))));
16   CONSTANT DEPTH_IACTS_BUFFER: NATURAL := FILTER_DEPTH * 2 *
     ↪  IFMAP_SIZE * IFMAP_SIZE;
17
18   TYPE iacts_buffer_type IS ARRAY(0 TO PE_COLUMNS - 1, 0 TO
     ↪  FILTER_PER_PE - 1) OF unsigned(DATA_WIDTH - 1 DOWNTO 0);
19   TYPE array_buffer IS ARRAY(0 TO 36 - 1) OF unsigned(DATA_WIDTH - 1
     ↪  DOWNTO 0);
20   TYPE iacts_mode_type IS (ENABLE_MEM, LOAD_IFMAP, PREPARE_IFMAP,
     ↪  CLEAN);
21   TYPE mode_psums_type IS (PREPARE_PSUMS, FETCH_PSUMS, WRITE_BACK,
     ↪  CLEAN, WRITE_OUT_PSUMS);
```

```
22
23    TYPE ifmap_address_type IS RECORD
24      shared_address : NATURAL;
25      shared_address_offset : NATURAL RANGE 0 TO 8;
26      address : NATURAL;
27    END RECORD;
28
29    TYPE psum_address_type IS RECORD
30      x : NATURAL RANGE 0 TO IFMAP_SIZE/PE_COLUMNS - 1;
31      y : NATURAL RANGE 0 TO IFMAP_SIZE - 1;
32      ofm : NATURAL RANGE 0 TO 3 - 1; --not used->because in parallel
        ↪  TODO! check
33    END RECORD;
34
35    TYPE ifmap_position_type IS RECORD
36      x : NATURAL RANGE 0 TO IFMAP_SIZE/PE_COLUMNS - 1;
37      y : NATURAL RANGE 0 TO IFMAP_SIZE - 1;
38      depth_pos : NATURAL RANGE 0 TO FILTER_DEPTH - 1;
39    END RECORD;
40
41    TYPE point IS RECORD
42      x : NATURAL RANGE 0 TO IFMAP_SIZE/PE_COLUMNS - 1;
43      y : NATURAL RANGE 0 TO IFMAP_SIZE - 1;
44    END RECORD;
45
46    FUNCTION to_buffer(v : STD_LOGIC_VECTOR(144 * 2 - 1 DOWNTO 0))
      ↪  RETURN array_buffer;
47
48  END PACKAGE;
49
50  PACKAGE BODY control_pck IS
51    FUNCTION to_buffer(v : STD_LOGIC_VECTOR(144 * 2 - 1 DOWNTO 0))
      ↪  RETURN array_buffer IS
52      VARIABLE buf : array_buffer := (OTHERS => (OTHERS => '0'));
53    BEGIN
54      FOR I IN 0 TO 36 - 1 LOOP
55        buf(I) := unsigned(v(DATA_WIDTH * (I + 1) - 1 DOWNTO DATA_WIDTH
          ↪  * I));
56      END LOOP;
57      RETURN buf;
58    END FUNCTION;
59
60  END PACKAGE BODY;
```

control/iacts_buffer.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
```

```
4  USE ieee.numeric_std.ALL;
5  USE work.core_pck.ALL;
6  USE work.control_pck.ALL;
7
8  USE work.ultra_ram_pck.ALL;
9  USE work.top_types_pck.ALL;
10
11 ENTITY iacts_buffer IS
12   GENERIC (
13     IFMAP_SIZE : NATURAL := 33;
14     IFMAPS_TO_PREPARE : NATURAL := 3;
15     AWIDTH : NATURAL := 18;
16     DEPTH : NATURAL := 32
17   );
18   PORT (
19     clk : IN STD_LOGIC;
20     reset : IN STD_LOGIC;
21     mode : IN iacts_mode_type;
22     values : IN ifmap_DRAM_type;
23     address : IN ifmap_position_type;
24     out_buffer : OUT iacts_buffer_type;
25     ifmaps_prepared : OUT STD_LOGIC
26   );
27 END ENTITY;
28
29 ARCHITECTURE arch OF iacts_buffer IS
30
31   CONSTANT NUM_COL : NATURAL := 9; -- 72/9 = 8
32   CONSTANT DWIDTH : NATURAL := 72 * 4;
33
34   --counters for addressing the ultraram
35   SIGNAL counter_a_nxt, counter_a, counter_b, counter_b_nxt :
     ↪  NATURAL;
36   --output buffer of ifmaps
37   SIGNAL out_buffer_nxt : iacts_buffer_type;
38   -- the state of the preperations
39   TYPE prepare_ifmap_state_type IS (PREP, WAIT_STATE, FIRST, LAST,
     ↪  FINISHED, STATIONARY);
40   SIGNAL prepare_ifmap_state, prepare_ifmap_state_nxt :
     ↪  prepare_ifmap_state_type;
41   -- URAM control signals
42   SIGNAL rsta : STD_LOGIC;
43   SIGNAL wea : STD_LOGIC_VECTOR(NUM_COL - 1 DOWNTO 0);
44   SIGNAL regcea : STD_LOGIC;
45   SIGNAL mem_ena : STD_LOGIC;
46   SIGNAL dina : STD_LOGIC_VECTOR(DWIDTH - 1 DOWNTO 0);
47   SIGNAL addra : STD_LOGIC_VECTOR(AWIDTH - 1 DOWNTO 0);
48   SIGNAL douta : STD_LOGIC_VECTOR(DWIDTH - 1 DOWNTO 0);
49   SIGNAL rstb : STD_LOGIC;
```

```
50    SIGNAL web : STD_LOGIC_VECTOR(NUM_COL - 1 DOWNTO 0);
51    SIGNAL regceb : STD_LOGIC;
52    SIGNAL mem_enb : STD_LOGIC;
53    SIGNAL dinb : STD_LOGIC_VECTOR(DWIDTH - 1 DOWNTO 0);
54    SIGNAL addrb : STD_LOGIC_VECTOR(AWIDTH - 1 DOWNTO 0);
55    SIGNAL doutb : STD_LOGIC_VECTOR(DWIDTH - 1 DOWNTO 0);
56
57    SIGNAL debug : array_buffer;
58
59    --forces wait until output from the URAM is valid
60    SIGNAL wait_counter, wait_counter_nxt : NATURAL RANGE 0 TO 4;
61    --write addresses for initializing the memory
62    SIGNAL write_addr, write_addr_nxt : NATURAL;
63    SIGNAL ifmap_counter, ifmap_counter_nxt : NATURAL RANGE 0 TO
      ↪  PE_COLUMNS - 1;
64    CONSTANT ULTRA_RAM_DWIDTH : NATURAL := 72 * 4;
65  BEGIN
66
67    ultra : xilinx_ultraram_true_dual_port_byte_write
68    GENERIC MAP(
69      AWIDTH => AWIDTH,
70      DWIDTH => ULTRA_RAM_DWIDTH,
71      NUM_COL => 9,
72      NBPIPE => 3,
73      DEPTH => DEPTH
74    )
75    PORT MAP(
76      clk => clk,
77      rsta => rsta,
78      wea => wea,
79      regcea => regcea,
80      mem_ena => mem_ena,
81      dina => dina,
82      addra => addra,
83      douta => douta,
84
85      rstb => rstb,
86      web => web,
87      regceb => regceb,
88      mem_enb => mem_enb,
89      dinb => dinb,
90      addrb => addrb,
91      doutb => doutb
92    );
93
94    sync : PROCESS (clk, reset)
95    BEGIN
96      IF reset = '0' THEN
97        prepare_ifmap_state <= PREP;
```

```
98        wait_counter <= 0;
99        out_buffer <= (OTHERS => (OTHERS => (OTHERS => '0')));
100       write_addr <= 0;
101       ifmap_counter <= 0;
102       counter_a <= 0;
103       counter_b <= 1;
104     ELSIF rising_edge(clk) THEN
105       out_buffer <= out_buffer_nxt;
106       prepare_ifmap_state <= prepare_ifmap_state_nxt;
107       write_addr <= write_addr_nxt;
108       wait_counter <= wait_counter_nxt;
109       ifmap_counter <= ifmap_counter_nxt;
110       counter_a <= counter_a_nxt;
111       counter_b <= counter_b_nxt;
112     END IF;
113   END PROCESS;
114   --prepares and control the ifmaps
115   in_out : PROCESS (ALL)
116     VARIABLE start_address : NATURAL;
117   BEGIN
118     rsta <= '0';
119     rstb <= '0';
120     IF reset = '0' THEN
121       rsta <= '1';
122       rstb <= '1';
123     END IF;
124
125     mem_ena <= '1';
126     mem_enb <= '1';
127     wea <= (OTHERS => '0');
128     web <= (OTHERS => '0');
129     regcea <= '0';
130     dina <= (OTHERS => '0');
131     dinb <= (OTHERS => '0');
132     addra <= (OTHERS => '0');
133     addrb <= (OTHERS => '0');
134     regceb <= '0';
135     ifmaps_prepared <= '0';
136     out_buffer_nxt <= out_buffer;
137     wait_counter_nxt <= 0;
138     counter_a_nxt <= counter_a;
139     counter_b_nxt <= counter_b;
140     ifmap_counter_nxt <= ifmap_counter;
141     --FOR I IN 0 TO 8 LOOP
142     --  debug <= to_buffer(douta);
143     --END LOOP;
144     write_addr_nxt <= write_addr;
145     prepare_ifmap_state_nxt <= prepare_ifmap_state;
146
```

```
147       CASE(mode) IS
148         WHEN ENABLE_MEM =>
149         WHEN LOAD_IFMAP =>
150
151         addra <= STD_LOGIC_VECTOR(to_unsigned(write_addr, AWIDTH));
152         addrb <= STD_LOGIC_VECTOR(to_unsigned(write_addr + 1, AWIDTH));
153         dina <= values.data(ULTRA_RAM_DWIDTH - 1 DOWNTO 0);
154         dinb <= values.data(ULTRA_RAM_DWIDTH * 2 - 1 DOWNTO
            ↪  ULTRA_RAM_DWIDTH);
155         IF values.valid = '1' THEN
156           wea <= (OTHERS => '1');
157           web <= (OTHERS => '1');
158           write_addr_nxt <= write_addr + 2;
159         END IF;
160
161         --prepares the ifmaps
162         WHEN PREPARE_IFMAP =>
163         regcea <= '1';
164         regceb <= '1';
165         start_address := address.x * PE_COLUMNS * 2 + address.y *
            ↪  IFMAP_SIZE * 2 + address.depth_pos * IFMAP_SIZE *
            ↪  IFMAP_SIZE * 2;--address.x*3*2 + address.y*11*3*2 +
            ↪  address.depth_pos*11*3*33*2;
166
167         addra <= STD_LOGIC_VECTOR(to_unsigned(start_address +
            ↪  counter_a, AWIDTH));
168         addrb <= STD_LOGIC_VECTOR(to_unsigned(start_address + counter_a
            ↪  + 1, AWIDTH));
169         counter_a_nxt <= counter_a + 2;
170
171         CASE(prepare_ifmap_state) IS
172
173           WHEN PREP =>
174           prepare_ifmap_state_nxt <= WAIT_STATE;
175
176           WHEN WAIT_STATE =>
177           wait_counter_nxt <= wait_counter + 1;
178           prepare_ifmap_state_nxt <= WAIT_STATE;
179           IF wait_counter = 3 THEN
180             prepare_ifmap_state_nxt <= STATIONARY;
181           END IF;
182
183           WHEN STATIONARY =>
184           --loops and waits until IFMAPS_TO_PREPARE ifmaps have been
              ↪  prepared
185           FOR I IN 0 TO 36 - 1 LOOP
186             out_buffer_nxt(ifmap_counter, I) <= to_buffer(douta)(I);
187             IF I + 36 < 64 THEN
```

```
188               out_buffer_nxt(ifmap_counter, I + 36) <=
                  ↪  to_buffer(doutb)(I);
189             END IF;
190           END LOOP;
191
192           IF ifmap_counter = IFMAPS_TO_PREPARE - 1 THEN
193             ifmap_counter_nxt <= 0;
194             prepare_ifmap_state_nxt <= FINISHED;
195           ELSE
196             ifmap_counter_nxt <= ifmap_counter + 1;
197           END IF;
198           WHEN FINISHED =>
199           ifmaps_prepared <= '1';
200           counter_a_nxt <= 0;
201           counter_b_nxt <= 1;
202           ifmap_counter_nxt <= 0;
203
204           WHEN OTHERS =>
205
206         END CASE;
207         WHEN CLEAN =>
208         out_buffer_nxt <= (OTHERS => (OTHERS => (OTHERS => '0')));
209         prepare_ifmap_state_nxt <= PREP;
210         counter_a_nxt <= 0;
211         ifmap_counter_nxt <= 0;
212       END CASE;
213     END PROCESS;
214
215   END ARCHITECTURE;
```

control/psums_buffer.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3
4   USE ieee.numeric_std.ALL;
5   USE work.core_pck.ALL;
6   USE work.control_pck.ALL;
7   USE work.ultra_ram_pck.ALL;
8   USE work.top_types_pck.ALL;
9
10  ENTITY psums_buffer IS
11    GENERIC (
12      ACC_WIDTH : NATURAL := 24;
13      MAX_OFMS : NATURAL := 3;
14      PARALLEL_OFMS : NATURAL := 4;
15      PE_COLUMNS : NATURAL := 3;
16      A_WIDTH : NATURAL := 9;
17      DWIDTH : NATURAL := 24 * 3 * 4;
```

```
18      DEPTH : NATURAL := 33 * 11
19    );
20
21    PORT (
22      clk : IN STD_LOGIC;
23      reset : IN STD_LOGIC;
24      mode : IN mode_psums_type;
25      address : IN point;
26      address_prev : IN point;
27      psums_ready : OUT STD_LOGIC;
28      psum_values_in : IN psum_array;
29      first_pass : IN STD_LOGIC;
30      out_buffer : OUT psum_array;
31      ofms_out : OUT ofms_out_type;
32      finished_writing_ofm : OUT STD_LOGIC
33    );
34
35  END ENTITY;
36
37  ARCHITECTURE arch OF psums_buffer IS
38
39    --TYPE mode_psums_type IS (WRITE_PSUMS, PREPARE_PSUMS, CLEAN);
40
41    --BRAM control
42    SIGNAL we : STD_LOGIC;
43    SIGNAL ena : STD_LOGIC;
44    SIGNAL raddr : STD_LOGIC_VECTOR(A_WIDTH - 1 DOWNTO 0);
45    SIGNAL waddr : STD_LOGIC_VECTOR(A_WIDTH - 1 DOWNTO 0);
46    SIGNAL din : STD_LOGIC_VECTOR(DWIDTH - 1 DOWNTO 0);
47    SIGNAL dout : STD_LOGIC_VECTOR(DWIDTH - 1 DOWNTO 0);
48
49    TYPE prepare_psums_state_type IS (WAIT_STATE, STATIONARY,
      ↪  FINISHED);
50    SIGNAL prepare_psums_state, prepare_psums_state_nxt :
      ↪  prepare_psums_state_type;
51    SIGNAL out_buffer_nxt : psum_array;
52    SIGNAL psums_reg_in_nxt, psums_reg_in : psum_array;
53    SIGNAL write_out_counter, write_out_counter_nxt : NATURAL;
54    --used for waiting until the output is valid (i.e. as dictated by
      ↪  in address)
55    CONSTANT MEM_DELAY : NATURAL := 2;
56    SIGNAL wait_counter, wait_counter_nxt : NATURAL RANGE 0 TO
      ↪  MEM_DELAY;
57    SIGNAL valid_ofm_nxt, valid_ofm : STD_LOGIC;
58
59  BEGIN
60
61    sync : PROCESS (clk, reset)
62    BEGIN
```

```
63      IF reset = '0' THEN
64        wait_counter <= 0;
65        prepare_psums_state <= WAIT_STATE;
66        out_buffer <= (OTHERS => (OTHERS => (OTHERS => '0')));
67        wait_counter <= 0;
68        psums_reg_in <= (OTHERS => (OTHERS => (OTHERS => '0')));
69        write_out_counter <= 0;
70        valid_ofm <= '0';
71      ELSIF rising_edge(clk) THEN
72        wait_counter <= wait_counter_nxt;
73        prepare_psums_state <= prepare_psums_state_nxt;
74        out_buffer <= out_buffer_nxt;
75        wait_counter <= wait_counter_nxt;
76        psums_reg_in <= psums_reg_in_nxt;
77        write_out_counter <= write_out_counter_nxt;
78        valid_ofm <= valid_ofm_nxt;
79      END IF;
80    END PROCESS;
81
82    rams_sdp_record_i : ENTITY work.rams_sdp_record
83      GENERIC MAP(
84        A_WID => A_WIDTH,
85        D_WID => DWIDTH,
86        DEPTH => DEPTH
87      )
88      PORT MAP(
89        clk => clk,
90        we => we,
91        ena => ena,
92        raddr => raddr,
93        waddr => waddr,
94        din => din,
95        dout => dout
96      );
97
98    state : PROCESS (ALL)
99    BEGIN
100     ena <= '1';
101     we <= '0';
102
103     din <= (OTHERS => '0');
104     out_buffer_nxt <= out_buffer;
105     prepare_psums_state_nxt <= prepare_psums_state;
106     wait_counter_nxt <= 0;
107     psums_reg_in_nxt <= psums_reg_in;
108     psums_ready <= '0';
109     finished_writing_ofm <= '0';
110     ofms_out.valid <= valid_ofm;
111     ofms_out.data <= (OTHERS => '0');
```

```
112       write_out_counter_nxt <= 0;
113       raddr <= (OTHERS => '0');
114       valid_ofm_nxt <= '0';
115       waddr <= (OTHERS => '0');
116
117       CASE(mode) IS
118
119         WHEN FETCH_PSUMS =>
120         psums_reg_in_nxt <= psum_values_in;
121
122         WHEN WRITE_OUT_PSUMS =>
123
124         raddr <= STD_LOGIC_VECTOR(to_unsigned(write_out_counter,
          ↪  raddr'length));
125         write_out_counter_nxt <= write_out_counter + 1;
126         valid_ofm_nxt <= '1';
127         IF write_out_counter > 0 THEN
128           ofms_out.data <= dout;
129           valid_ofm_nxt <= '1';
130         ELSE
131           ofms_out.valid <= '0';
132         END IF;
133         IF write_out_counter > DEPTH - 1 THEN
134           raddr <= STD_LOGIC_VECTOR(to_unsigned(0, raddr'length));
135           valid_ofm_nxt <= '0';
136         END IF;
137         IF write_out_counter = DEPTH + MEM_DELAY + 1 THEN
138           write_out_counter_nxt <= 0;
139           finished_writing_ofm <= '1';
140         END IF;
141
142         WHEN PREPARE_PSUMS =>
143
144         raddr <= STD_LOGIC_VECTOR(to_unsigned(address.y *
          ↪  IFMAP_SIZE/PE_COLUMNS + address.x, raddr'length));
145         waddr <= STD_LOGIC_VECTOR(to_unsigned(address_prev.y *
          ↪  IFMAP_SIZE/PE_COLUMNS + address_prev.x, waddr'length));
146
147         -- write the psum reg that has first been fetched back to
          ↪  memory
148         FOR ofm IN 0 TO PARALLEL_OFMS - 1 LOOP
149           we <= '1';
150           FOR I IN 0 TO PE_COLUMNS - 1 LOOP
151             din(((ofm * PE_COLUMNS + I) + 1) * ACC_DATA_WIDTH - 1
                ↪  DOWNTO (ofm * PE_COLUMNS + I) * ACC_DATA_WIDTH) <=
                ↪  STD_LOGIC_VECTOR(signed(psums_reg_in(ofm, I)));
152           END LOOP;
153         END LOOP;
154
```

```
155         CASE (prepare_psums_state) IS
156           WHEN WAIT_STATE =>
157
158             IF wait_counter = MEM_DELAY THEN
159               prepare_psums_state_nxt <= STATIONARY;
160               wait_counter_nxt <= 0;
161             ELSE
162               wait_counter_nxt <= wait_counter + 1;
163             END IF;
164
165           WHEN STATIONARY =>
166             IF first_pass = '1' THEN
167               out_buffer_nxt <= (OTHERS => (OTHERS => (OTHERS =>
                  ↪  '0')));
168             ELSE
169               FOR ofm IN 0 TO PARALLEL_OFMS - 1 LOOP
170                 FOR x IN 0 TO PE_COLUMNS - 1 LOOP
171                   out_buffer_nxt(ofm, x) <= signed(dout(((ofm *
                      ↪  PE_COLUMNS + x) + 1) * ACC_DATA_WIDTH - 1 DOWNTO
                      ↪  (ofm * PE_COLUMNS + x) * ACC_DATA_WIDTH));
172                 END LOOP;
173               END LOOP;
174             END IF;
175             prepare_psums_state_nxt <= FINISHED;
176           WHEN FINISHED =>
177             psums_ready <= '1';
178             wait_counter_nxt <= 0;
179         END CASE;
180
181         WHEN WRITE_BACK =>
182
183         WHEN CLEAN =>
184         write_out_counter_nxt <= 0;
185         prepare_psums_state_nxt <= WAIT_STATE;
186         wait_counter_nxt <= 0;
187       END CASE;
188     END PROCESS;
189
190 END ARCHITECTURE;
```

control/state_calc.vhd

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE ieee.std_logic_misc.ALL;
5 USE work.core_pck.ALL;
6 USE work.control_pck.ALL;
7 USE work.state_calc_pkg.ALL;
```

```
8
9  ENTITY state_calc IS
10   GENERIC (
11     PARALLEL_OFMS : NATURAL := 3;
12     MAX_OFMS : NATURAL := 255;
13     FILTER_DEPTH : NATURAL := 32;
14     FILTER_VALUES : NATURAL := 64;
15     MAX_RATE : NATURAL := 3
16   );
17   PORT (
18     clk, reset : IN STD_LOGIC;
19     new_ifmaps : IN STD_LOGIC;
20     need_kernel : OUT STD_LOGIC;
21     first_pass : OUT STD_LOGIC;
22     write_out_ofms_out : OUT STD_LOGIC;
23     ifmap_position : OUT ifmap_position_type;
24     psums_position : OUT point;
25     psums_position_prev : OUT point;
26     psum_prev_address, finished : OUT STD_LOGIC
27   );
28 END ENTITY;
29
30 ARCHITECTURE arch OF state_calc IS
31
32   --the calculated ifmap/psums positions
33   SIGNAL ifmap_position_nxt, ifmap_position_prev_nxt :
     ↪  ifmap_position_type;
34   SIGNAL psums_position_prev_nxt, psums_position_curr,
     ↪  psums_position_curr_nxt : point;
35   --detect if new ifmap has been writen i.e. advance state
36   SIGNAL new_ifmaps_reg, new_ifmaps_reg_nxt, finished_nxt,
     ↪  need_kernel_nxt : STD_LOGIC;
37   --rate multiplier, only 1 currently supported
38   SIGNAL rate, rate_nxt : NATURAL RANGE 1 TO 3;
39   --the kernel position
40   SIGNAL kernel, kernel_nxt : NATURAL RANGE 0 TO 8;
41   --the ofm position
42   SIGNAL ofm, ofm_nxt : NATURAL RANGE 0 TO MAX_OFMS - 1 +
     ↪  PARALLEL_OFMS;
43   SIGNAL first_pass_nxt : STD_LOGIC;
44   SIGNAL write_out_ofms_nxt, write_out_ofms : STD_LOGIC_VECTOR(1
     ↪  DOWNTO 0);
45   SIGNAL finished_curr, finished_curr_nxt : STD_LOGIC;
46   SIGNAL write_out_ofms_reg : STD_LOGIC;
47 BEGIN
48   sync : PROCESS (clk, reset)
49   BEGIN
50     IF reset = '0' THEN
51       rate <= 1;
```

```
52        kernel <= 4;
53        ifmap_position.x <= 0;
54        ifmap_position.y <= 0;
55        ifmap_position.depth_pos <= 0;
56        ofm <= 0;
57        finished <= '0';
58        need_kernel <= '1';
59        first_pass <= '1';
60        psums_position_prev <= (0, 0);
61        psums_position_curr <= (0, 0);
62        write_out_ofms <= (OTHERS => '0');
63        finished_curr <= '0';
64      ELSIF rising_edge(clk) THEN
65        ifmap_position <= ifmap_position_nxt;
66        new_ifmaps_reg <= new_ifmaps_reg_nxt;
67        rate <= rate_nxt;
68        first_pass <= first_pass_nxt;
69        kernel <= kernel_nxt;
70        ofm <= ofm_nxt;
71        finished <= finished_nxt;
72        need_kernel <= need_kernel_nxt;
73        psums_position_curr <= psums_position_curr_nxt;
74        psums_position_prev <= psums_position_prev_nxt;
75        write_out_ofms <= write_out_ofms_nxt;
76        finished_curr <= finished_curr_nxt;
77        write_out_ofms_reg <= write_out_ofms_out;
78      END IF;
79    END PROCESS;
80
81    psums_pos : PROCESS (ALL)
82    BEGIN
83      psums_position_prev_nxt <= psums_position_prev;
84      psums_position.x <= ifmap_position.x;
85      psums_position.y <= ifmap_position.y;
86      finished_nxt <= finished;
87      write_out_ofms_out <= write_out_ofms_reg;
88      psums_position_curr_nxt <= psums_position_curr;
89
90      IF kernel MOD 3 = 0 THEN
91        psums_position.x <= ifmap_position.x + rate *
          ↪  DILATION_RATE/PE_COLUMNS;
92      ELSIF kernel MOD 3 = 2 THEN
93        psums_position.x <= ifmap_position.x - rate *
          ↪  DILATION_RATE/PE_COLUMNS;
94      END IF;
95
96      IF kernel < 3 THEN
97        psums_position.y <= ifmap_position.y + rate * DILATION_RATE;
98      ELSIF kernel > 5 THEN
```

```
99        psums_position.y <= ifmap_position.y - rate * DILATION_RATE;
100     END IF;
101     IF new_ifmaps = '1' AND new_ifmaps_reg = '0' THEN
102       psums_position_curr_nxt <= psums_position;
103       write_out_ofms_out <= write_out_ofms(1);
104       finished_nxt <= finished_curr;
105     END IF;
106     IF new_ifmaps = '1' AND new_ifmaps_reg = '0' THEN
107       psums_position_prev_nxt <= psums_position_curr;
108     END IF;
109   END PROCESS;
110
111   state : PROCESS (ALL)
112     VARIABLE start_point, end_point : point;
113     VARIABLE kernel_var : NATURAL RANGE 0 TO 8;
114   BEGIN
115     ifmap_position_nxt <= ifmap_position;
116     rate_nxt <= rate;
117     finished_curr_nxt <= finished_curr;
118
119     kernel_nxt <= kernel;
120     ifmap_position_nxt.x <= ifmap_position.x;
121     ifmap_position_nxt.y <= ifmap_position.y;
122     kernel_var := kernel;
123     new_ifmaps_reg_nxt <= new_ifmaps;
124     need_kernel_nxt <= need_kernel;
125
126     first_pass_nxt <= first_pass;
127     write_out_ofms_nxt(1) <= write_out_ofms(1);
128     ofm_nxt <= ofm;
129     IF new_ifmaps = '1' AND new_ifmaps_reg = '0' THEN
130       write_out_ofms_nxt(1) <= '0';
131       --calc new ifmap_address
132       need_kernel_nxt <= '0';
133       start_end_point_calc(kernel_var, rate, start_point, end_point);
134       IF end_point.x = ifmap_position.x THEN
135         IF end_point.y = ifmap_position.y THEN
136           ifmap_position_nxt.x <= start_point.x;
137           ifmap_position_nxt.y <= start_point.y;
138           need_kernel_nxt <= '1';
139           first_pass_nxt <= '0';
140           IF kernel = 3 THEN --completly reset the ifmap
141             ifmap_position_nxt.x <= 0;
142             ifmap_position_nxt.y <= 0;
143             kernel_nxt <= 4;
144             IF ifmap_position.depth_pos = FILTER_DEPTH - 1 THEN
145               write_out_ofms_nxt(1) <= '1';
146               ifmap_position_nxt.depth_pos <= 0;
147               first_pass_nxt <= '1';
```

```
148                 IF ofm + PARALLEL_OFMS > MAX_OFMS - 1 THEN
149
150                   IF rate = MAX_RATE THEN
151                     finished_curr_nxt <= '1';
152
153                   ELSE
154                     rate_nxt <= rate + 1;
155                     ofm_nxt <= 0;
156                   END IF;
157                 ELSE
158                   ofm_nxt <= ofm + PARALLEL_OFMS;
159                 END IF;
160               ELSE
161                 ifmap_position_nxt.depth_pos <=
                    ↪  ifmap_position.depth_pos + 1;
162               END IF;
163             ELSE
164               IF kernel_var = 8 THEN
165                 kernel_var := 0;
166               ELSE
167                 kernel_var := kernel_var + 1;
168               END IF;
169
170               start_end_point_calc(kernel_var, rate, start_point,
                  ↪  end_point);
171               ifmap_position_nxt.x <= start_point.x;
172               ifmap_position_nxt.y <= start_point.y;
173               kernel_nxt <= kernel_var;
174
175             END IF;
176           ELSE
177             ifmap_position_nxt.x <= start_point.x;
178             ifmap_position_nxt.y <= ifmap_position.y + 1;
179           END IF;
180         ELSE
181           ifmap_position_nxt.x <= ifmap_position.x + 1;
182         END IF;
183       END IF;
184     END PROCESS;
185   END ARCHITECTURE;
```

control/state_calc_pkg.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL; --do I need this?
3   USE ieee.numeric_std.ALL;
4
5   USE work.core_pck.ALL;
6   USE work.control_pck.ALL;
```

```
7
8  PACKAGE state_calc_pkg IS
9    PROCEDURE start_end_point_calc(VARIABLE kernel : IN NATURAL RANGE 0
     ↪  TO 8;
10   SIGNAL rate : IN NATURAL RANGE 1 TO 3;
11   VARIABLE start_point, end_point : OUT point);
12 END PACKAGE;
13
14 PACKAGE BODY state_calc_pkg IS
15   PROCEDURE start_end_point_calc(VARIABLE kernel : IN NATURAL RANGE 0
     ↪  TO 8;
16   SIGNAL rate : IN NATURAL RANGE 1 TO 3;
17   VARIABLE start_point, end_point : OUT point) IS
18
19 BEGIN
20   IF kernel MOD 3 = 0 THEN
21     start_point.x := 0;
22     end_point.x := IFMAP_SIZE/PE_COLUMNS - 1 - rate *
       ↪  (DILATION_RATE/PE_COLUMNS);
23   ELSIF kernel MOD 3 = 1 THEN
24     start_point.x := 0;
25     end_point.x := IFMAP_SIZE/PE_COLUMNS - 1;
26   ELSE
27     start_point.x := rate * (DILATION_RATE/PE_COLUMNS);
28     end_point.x := IFMAP_SIZE/PE_COLUMNS - 1;
29   END IF;
30
31   IF kernel < 3 THEN
32     start_point.y := 0;
33     end_point.y := IFMAP_SIZE - 1 - rate * DILATION_RATE;
34   ELSIF kernel < 6 THEN
35     start_point.y := 0;
36     end_point.y := IFMAP_SIZE - 1;
37   ELSE
38     start_point.y := rate * DILATION_RATE;
39     end_point.y := IFMAP_SIZE - 1;
40   END IF;
41 END PROCEDURE;
42
43 END PACKAGE BODY;
```

control/ultra_ram.vhd

```
1  --  Taken from the Xilinx language VHDL examples!
2  --  Xilinx UltraRAM True Dual Port Mode with Byte-write.  This code
   ↪  implements
3  --  a parameterizable UltraRAM block with write/read on both ports in
4  --  No change behavior on both the ports . The behavior of this RAM
   ↪  is
```

```
5  --  when data is written, the output of RAM is unchanged w.r.t each
   ↪  port.
6  --  Only when write is inactive data corresponding to the address is
7  --  presented on the output port.
8  --
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.numeric_std.all;
12 entity xilinx_ultraram_true_dual_port_byte_write is
13 generic (
14           AWIDTH : integer := 19;  -- Address Width
15           DWIDTH : integer := 72;  -- Data Width
16           NUM_COL : integer := 9;  --72/NM_COL = Byte
17           NBPIPE : integer := 3;    -- Number of pipeline Registers
18           DEPTH: integer := 33*33*8*32
19         );
20 port     (
21           clk : in std_logic;                                    --
             ↪  Clock
22           -- Port A
23           rsta : in std_logic;                                    --
             ↪  Reset
24           wea : in std_logic_vector(NUM_COL-1 downto 0);          --
             ↪  Write Enable
25           regcea : in std_logic;                                  --
             ↪  Output Register Enable
26           mem_ena : in std_logic;                                 --
             ↪  Memory Enable
27           dina : in std_logic_vector(DWIDTH-1 downto 0);          --
             ↪  Data Input
28           addra : in std_logic_vector(AWIDTH-1 downto 0);         --
             ↪  Address Input
29           douta : out std_logic_vector(DWIDTH-1 downto 0);        --
             ↪  Data Output
30           -- Port B
31           rstb : in std_logic;                                    --
             ↪  Reset
32           web : in std_logic_vector(NUM_COL-1 downto 0);          --
             ↪  Write Enable
33           regceb : in std_logic;                                  --
             ↪  Output Register Enable
34           mem_enb : in std_logic;                                 --
             ↪  Memory Enable
35           dinb : in std_logic_vector(DWIDTH-1 downto 0);          --
             ↪  Data Input
36           addrb : in std_logic_vector(AWIDTH-1 downto 0);         --
             ↪  Address Input
37           doutb : out std_logic_vector(DWIDTH-1 downto 0)         --
             ↪  Data Output
```

```
38          );
39  end xilinx_ultraram_true_dual_port_byte_write;
40
41  architecture rtl of xilinx_ultraram_true_dual_port_byte_write is
42
43  constant C_AWIDTH : integer := AWIDTH;
44  constant C_DWIDTH : integer := DWIDTH;
45  constant C_NBPIPE : integer := NBPIPE;
46  constant CWIDTH : integer := DWIDTH/NUM_COL;
47  -- Internal Signals
48  type mem_t is array(natural range<>) of std_logic_vector(C_DWIDTH-1
    ↪  downto 0);
49  type pipe_data_t is array(natural range<>) of
    ↪  std_logic_vector(C_DWIDTH-1 downto 0);
50  type pipe_en_t is array(natural range<>) of std_logic;
51
52  shared variable mem : mem_t(2**C_AWIDTH-1 downto 0);
    ↪  -- Memory Declaration
53  signal memrega : std_logic_vector(C_DWIDTH-1 downto 0);
54  signal mem_pipe_rega : pipe_data_t(C_NBPIPE-1 downto 0);     --
    ↪  Pipelines for memory
55  signal mem_en_pipe_rega : pipe_en_t(C_NBPIPE downto 0);      --
    ↪  Pipelines for memory enable
56
57  signal memregb : std_logic_vector(C_DWIDTH-1 downto 0);
58  signal mem_pipe_regb : pipe_data_t(C_NBPIPE-1 downto 0);     --
    ↪  Pipelines for memory
59  signal mem_en_pipe_regb : pipe_en_t(C_NBPIPE downto 0);      --
    ↪  Pipelines for memory enable
60
61  constant zeros : std_logic_vector(NUM_COL-1 downto 0) := (others =>
    ↪  '0');
62  attribute ram_style : string;
63  attribute ram_style of mem : variable is "ultra";
64
65  begin
66
67  -- RAM : Read has one latency, Write has one latency as well.
68  process(clk)
69  begin
70    if(clk'event and clk='1')then
71      if(mem_ena = '1') then
72      for i in 0 to NUM_COL-1 loop
73        if(wea(i) = '1') then
74          mem(to_integer(unsigned(addra)))((i+1)*CWIDTH-1 downto
            ↪  i*CWIDTH) := dina((i+1)*CWIDTH-1 downto i*CWIDTH);
75        end if;
76       end loop;
77      end if;
```

```
78    end if;
79  end process;
80
81  process(clk)
82  begin
83   if(clk'event and clk='1')then
84    if(mem_ena = '1') then
85      if( wea = zeros) then
86        memrega <= mem(to_integer(unsigned(addra)));
87      end if;
88    end if;
89   end if;
90  end process;
91  -- The enable of the RAM goes through a pipeline to produce a
92  -- series of pipelined enable signals required to control the data
93  -- pipeline.
94  process(clk)
95  begin
96    if(clk'event and clk = '1') then
97      mem_en_pipe_rega(0) <= mem_ena;
98      for i in 0 to C_NBPIPE-1 loop
99        mem_en_pipe_rega(i+1) <= mem_en_pipe_rega(i);
100     end loop;
101   end if;
102 end process;
103
104 -- RAM output data goes through a pipeline.
105 process(clk)
106 begin
107   if(clk'event and clk = '1') then
108     if(mem_en_pipe_rega(0) = '1') then
109       mem_pipe_rega(0) <= memrega;
110     end if;
111     for i in 0 to C_NBPIPE-2 loop
112       if(mem_en_pipe_rega(i+1) = '1') then
113         mem_pipe_rega(i+1) <= mem_pipe_rega(i);
114       end if;
115     end loop;
116   end if;
117 end process;
118
119 -- Final output register gives user the option to add a reset and
120 -- an additional enable signal just for the data ouptut
121
122 process(clk)
123 begin
124   if(clk'event and clk = '1') then
125     if(rsta = '1') then
126       douta <= (others => '0');
```

```
127     elsif(mem_en_pipe_rega(C_NBPIPE) = '1' and regcea = '1' ) then
128       douta <= mem_pipe_rega(C_NBPIPE-1);
129     end if;
130   end if;
131 end process;
132
133
134 -- RAM : Read has one latency, Write has one latency as well.
135 process(clk)
136 begin
137   if(clk'event and clk='1')then
138     if(mem_enb = '1') then
139      for i in 0 to NUM_COL-1 loop
140       if(web(i) = '1') then
141         mem(to_integer(unsigned(addrb)))((i+1)*CWIDTH-1 downto
          ↪  i*CWIDTH) := dinb((i+1)*CWIDTH-1 downto i*CWIDTH);
142       end if;
143      end loop;
144
145
146     end if;
147
148   end if;
149
150
151 end process;
152
153 process(clk)
154 begin
155  if(clk'event and clk='1')then
156   if(mem_enb = '1') then
157     if(web = zeros) then
158       memregb <= mem(to_integer(unsigned(addrb)));
159     end if;
160   end if;
161  end if;
162 end process;
163
164 -- The enable of the RAM goes through a pipeline to produce a
165 -- series of pipelined enable signals required to control the data
166 -- pipeline.
167 process(clk)
168 begin
169   if(clk'event and clk = '1') then
170     mem_en_pipe_regb(0) <= mem_enb;
171     for i in 0 to C_NBPIPE-1 loop
172       mem_en_pipe_regb(i+1) <= mem_en_pipe_regb(i);
173     end loop;
174   end if;
```

```
175 end process;
176
177 -- RAM output data goes through a pipeline.
178 process(clk)
179 begin
180   if(clk'event and clk = '1') then
181     if(mem_en_pipe_regb(0) = '1') then
182       mem_pipe_regb(0) <= memregb;
183     end if;
184     for i in 0 to C_NBPIPE-2 loop
185       if(mem_en_pipe_regb(i+1) = '1') then
186         mem_pipe_regb(i+1) <= mem_pipe_regb(i);
187       end if;
188     end loop;
189   end if;
190 end process;
191
192 -- Final output register gives user the option to add a reset and
193 -- an additional enable signal just for the data ouptut
194
195 process(clk)
196 begin
197   if(clk'event and clk = '1') then
198     if(rstb = '1') then
199       doutb <= (others => '0');
200     elsif(mem_en_pipe_regb(C_NBPIPE) = '1' and regceb = '1' ) then
201       doutb <= mem_pipe_regb(C_NBPIPE-1);
202     end if;
203   end if;
204 end process;
205
206
207
208
209 end rtl;
210
211
212
213 -- The following is an instantiation template for
    ↪  xilinx_ultraram_true_dual_port_byte_write
214 -- Component Declaration
215 -- Uncomment the below component declaration when using
216 -- component xilinx_ultraram_true_dual_port_byte_write
217 --generic (
218 --          AWIDTH : integer := 12;  -- Address Width
219 --          DWIDTH : integer := 72;  -- Data Width
220 --          NUM_COL : integer := 9;  -- Number of columns
221 --          NBPIPE : integer := 3    -- Number of pipeline Registers
222 --        );
```

```
223 --port    (
224 --         clk : in std_logic;                                    --
    ↪  Clock
225
226 --         rsta : in std_logic;                                    --
    ↪  Reset
227 --         wea : in std_logic_vector(NUM_COL-1 downto 0);         --
    ↪  Write Enable
228 --         regcea : in std_logic;                                  --
    ↪  Output Register Enable
229 --         mem_ena : in std_logic;                                 --
    ↪  Memory Enable
230 --         dina : in std_logic_vector(DWIDTH-1 downto 0);         --
    ↪  Data Input
231 --         addra : in std_logic_vector(AWIDTH-1 downto 0);        --
    ↪  Address Input
232 --         douta : out std_logic_vector(DWIDTH-1 downto 0);        --
    ↪  Data Output
233
234 --         rstb : in std_logic;                                    --
    ↪  Reset
235 --         web : in std_logic_vector(NUM_COL-1 downto 0);         --
    ↪  Write Enable
236 --         regceb : in std_logic;                                  --
    ↪  Output Register Enable
237 --         mem_enb : in std_logic;                                 --
    ↪  Memory Enable
238 --         dinb : in std_logic_vector(DWIDTH-1 downto 0);         --
    ↪  Data Input
239 --         addrb : in std_logic_vector(AWIDTH-1 downto 0);        --
    ↪  Address Input
240 --         doutb : out std_logic_vector(DWIDTH-1 downto 0)        --
    ↪  Data Output
241
242 --        );
243 -- end component;
244 -- Instantiation
245 -- Uncomment the below component declaration when using
246 -- <your_instance_name> : xilinx_ultraram_true_dual_port_byte_write
247 -- generic map (
248 --      AWIDTH => AWIDTH,
249 --      DWIDTH => DWIDTH,
250 --      NUM_COL => NUM_COL,
251 --      NBPIPE => NBPIPE
252 --       )
253 -- port map (
254 --         clk => clk,
255 --         rsta => rsta,
256 --         wea => wea,
```

```
257 --          regcea => regcea,
258 --          mem_ena => mem_ena,
259 --          dina => dina,
260 --          addra => addra,
261 --          douta => douta,
262 --
263 --          rstb => rstb,
264 --          web => web,
265 --          regceb => regceb,
266 --          mem_enb => mem_enb,
267 --          dinb => dinb,
268 --          addrb => addrb,
269 --          doutb => doutb
270 --          );
```

control/ultra_ram_pkg.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  use ieee.numeric_std.all;
5
6
7  package ultra_ram_pck is
8    component xilinx_ultraram_true_dual_port_byte_write
9    generic (
10             AWIDTH : integer := 12;  -- Address Width
11             DWIDTH : integer := 72;  -- Data Width
12             NUM_COL : integer := 9;  -- Number of columns
13             NBPIPE : integer := 3;    -- Number of pipeline Registers
14             DEPTH : integer := 32
15            );
16    port     (
17             clk : in std_logic;                                    --
               ↪ Clock
18
19             rsta : in std_logic;                                    --
               ↪ Reset
20             wea : in std_logic_vector(NUM_COL-1 downto 0);          --
               ↪ Write Enable
21             regcea : in std_logic;                                  --
               ↪ Output Register Enable
22             mem_ena : in std_logic;                                 --
               ↪ Memory Enable
23             dina : in std_logic_vector(DWIDTH-1 downto 0);          --
               ↪ Data Input
24             addra : in std_logic_vector(AWIDTH-1 downto 0);         --
               ↪ Address Input
```

```
25              douta : out std_logic_vector(DWIDTH-1 downto 0);        --
                ↪  Data Output
26
27              rstb : in std_logic;                                    --
                ↪  Reset
28              web : in std_logic_vector(NUM_COL-1 downto 0);          --
                ↪  Write Enable
29              regceb : in std_logic;                                  --
                ↪  Output Register Enable
30              mem_enb : in std_logic;                                 --
                ↪  Memory Enable
31              dinb : in std_logic_vector(DWIDTH-1 downto 0);          --
                ↪  Data Input
32              addrb : in std_logic_vector(AWIDTH-1 downto 0);         --
                ↪  Address Input
33              doutb : out std_logic_vector(DWIDTH-1 downto 0)         --
                ↪  Data Output
34
35            );
36          end component;
37
38  end package;
```

ofm/block_ram.vhd

```
1   -- Ram Inference Example using Records (Simple Dual port)
2   -- File:rams_sdp_record.vhd
3   -- taken from xilinx language example designs
4
5   LIBRARY ieee;
6   USE ieee.std_logic_1164.ALL;
7
8   LIBRARY ieee;
9   USE ieee.std_logic_1164.ALL;
10  USE ieee.numeric_std.ALL;
11
12  ENTITY rams_sdp_record IS GENERIC (
13    A_WID : INTEGER := 17;
14    D_WID : INTEGER := 36;
15    DEPTH : INTEGER := 33 * 33
16  );
17  PORT (
18    clk : IN STD_LOGIC;
19    we : IN STD_LOGIC;
20    ena : IN STD_LOGIC;
21    raddr : IN STD_LOGIC_VECTOR(A_WID - 1 DOWNTO 0);
22    waddr : IN STD_LOGIC_VECTOR(A_WID - 1 DOWNTO 0);
23    din : IN STD_LOGIC_VECTOR(D_WID - 1 DOWNTO 0);
24    dout : OUT STD_LOGIC_VECTOR(D_WID - 1 DOWNTO 0)
```

```
);
END rams_sdp_record;

ARCHITECTURE arch OF rams_sdp_record IS
  TYPE mem_t IS ARRAY(INTEGER RANGE <>) OF STD_LOGIC_VECTOR(D_WID - 1
  ↪ DOWNTO 0);
  SIGNAL mem : mem_t(DEPTH - 1 DOWNTO 0) := (OTHERS => (OTHERS =>
  ↪ '0'));
BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk'event AND clk = '1') THEN
      IF (ena = '1') THEN
        IF (we = '1') THEN
          mem(to_integer(unsigned(waddr))) <= din;
        END IF;
      END IF;
    END IF;
  END PROCESS;

  PROCESS (clk)
  BEGIN
    IF (clk'event AND clk = '1') THEN
      IF (ena = '1') THEN
        dout <= mem(to_integer(unsigned(raddr)));
      END IF;
    END IF;
  END PROCESS;

END arch;
```

ofm/block_ram_pkg.vhd

```
-- Taken form Xilinx language example designs
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

PACKAGE block_ram_pkg IS

  COMPONENT block_ram IS
```

```
17      GENERIC (
18        ADDR_WIDTH : NATURAL; -- bitwidth of address
19        DATA_WIDTH : NATURAL -- bitwidth of data
20      );
21      PORT (
22        clk : IN STD_LOGIC;
23        addr_a : IN STD_LOGIC_VECTOR(ADDR_WIDTH - 1 DOWNTO 0); -- port
          ↪ a address
24        addr_b : IN STD_LOGIC_VECTOR(ADDR_WIDTH - 1 DOWNTO 0); -- port
          ↪ b address
25        din_a : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0); -- port a
          ↪ write data
26        din_b : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0); -- port b
          ↪ write data
27        en_a : IN STD_LOGIC; -- port a enable
28        en_b : IN STD_LOGIC; -- port b enable
29        we_a : IN STD_LOGIC; -- port a write-enable
30        we_b : IN STD_LOGIC; -- port b write-enable
31        dout_a : OUT STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0); -- port
          ↪ a read data
32        dout_b : OUT STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0) -- port
          ↪ b read data
33      );
34    END COMPONENT block_ram;
35
36  END block_ram_pkg;
```

ofm/init_file_bram.vhd

```
1   -- Initializing Block RAM from external data file
2   -- File: rams_init_file.vhd
3   -- adapted from xilinx example designs
4   LIBRARY ieee;
5   USE ieee.std_logic_1164.ALL;
6   USE ieee.numeric_std.ALL;
7   USE std.textio.ALL;
8
9   ENTITY rams_init_file IS
10    GENERIC (
11      FILENAME : STRING := "filename.data";
12      ADDRW : NATURAL := 8;
13      DATAW : NATURAL := 8;
14      DEPTH : NATURAL := 255 * 3
15    );
16    PORT (
17      clk : IN STD_LOGIC;
18      addr : IN STD_LOGIC_VECTOR(ADDRW - 1 DOWNTO 0);
19      dout : OUT STD_LOGIC_VECTOR(DATAW - 1 DOWNTO 0)
20    );
```

```
21 END rams_init_file;
22
23 ARCHITECTURE syn OF rams_init_file IS
24   TYPE RamType IS ARRAY (0 TO DEPTH - 1) OF bit_vector(DATAW - 1
     ↪  DOWNTO 0);
25
26   IMPURE FUNCTION InitRamFromFile(RamFileName : IN STRING) RETURN
     ↪  RamType IS
27     FILE RamFile : text OPEN read_mode IS RamFileName;
28     VARIABLE RamFileLine : line;
29     VARIABLE RAM : RamType;
30   BEGIN
31     FOR I IN RamType'RANGE LOOP
32       readline(RamFile, RamFileLine);
33       read(RamFileLine, RAM(I));
34     END LOOP;
35     RETURN RAM;
36   END FUNCTION;
37
38   SIGNAL RAM : RamType := InitRamFromFile(FILENAME);
39 BEGIN
40   PROCESS (clk)
41   BEGIN
42     IF clk'event AND clk = '1' THEN
43       dout <= to_stdlogicvector(RAM(to_integer(unsigned(addr))));
44     END IF;
45   END PROCESS;
46 END syn;
```

ofm/ofm_unit.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  USE ieee.numeric_std.ALL;
5  USE IEEE.math_real.ALL;
6  USE work.core_pck.ALL;
7  USE work.control_pck.ALL;
8  USE work.top_types_pck.ALL;
9  ENTITY ofms_unit IS
10   GENERIC (
11     PARALLEL_OFMS : NATURAL := 3;
12     MAX_OFMS : NATURAL := 255;
13     MAX_RATE : NATURAL := 3;
14     PE_COLUMNS : NATURAL := 3;
15     OFM_REQUANT : NATURAL := 62
16   );
17   PORT (
18     clk : IN STD_LOGIC;
```

```
19      reset : IN STD_LOGIC;
20      ofms_in : IN ofms_out_type;
21      from_uart : IN from_uart_type;
22      to_uart : OUT to_uart_type --to modify
23    );
24  END ENTITY;
25  -- this architecture is not well implemented (it is correct however)
    ↪  and propably should be split into multiple entities
26  -- Roughly the entity takes in the finished ofms, subsequently it
    ↪  puts them into the requant pipeline
27  -- the results of the requant pipeline are saved in BRAM until all
    ↪  ofms have been saved.
28  -- Once all have been saved they are communicated to the uart unit
    ↪  and written out by it.
29  ARCHITECTURE arch OF ofms_unit IS
30
31    CONSTANT D_WID : NATURAL := PARALLEL_OFMS * PE_COLUMNS *
      ↪  DATA_WIDTH;
32
33    --first stage compute real values
34    --second stage write the result to the BRAMs
35    SIGNAL dout_scale : STD_LOGIC_VECTOR(32 - 1 DOWNTO 0);
36
37    -- basic values
38    CONSTANT ofm_quant : NATURAL := OFM_REQUANT;
39    CONSTANT OFM_SIZE_IFMAP : NATURAL := IFMAP_SIZE *
      ↪  (IFMAP_SIZE/PE_COLUMNS);
40    CONSTANT MAX_ADDR_DATA : NATURAL := (MAX_OFMs/PARALLEL_OFMS) *
      ↪  MAX_RATE * OFM_SIZE_IFMAP; --3
41    CONSTANT MAX_ADDR_ITER : NATURAL := PARALLEL_OFMS * MAX_RATE *
      ↪  OFM_SIZE_IFMAP;
42    CONSTANT VAlUES_WIDTH : NATURAL := PARALLEL_OFMS * PE_COLUMNS;
43    CONSTANT A_WID : NATURAL :=
      ↪  INTEGER(ceil(log2(real(MAX_ADDR_DATA))));
44    CONSTANT PARALLEL_DATA : NATURAL := PARALLEL_OFMS * PE_COLUMNS;
45    -- ofm memory control
46    SIGNAL we : STD_LOGIC;
47    SIGNAL ena : STD_LOGIC;
48    SIGNAL raddr : STD_LOGIC_VECTOR(A_WID - 1 DOWNTO 0);
49    SIGNAL waddr : STD_LOGIC_VECTOR(A_WID - 1 DOWNTO 0);
50    SIGNAL din : STD_LOGIC_VECTOR(PARALLEL_OFMS * PE_COLUMNS *
      ↪  DATA_WIDTH - 1 DOWNTO 0);
51    SIGNAL dout : STD_LOGIC_VECTOR(PARALLEL_OFMS * PE_COLUMNS *
      ↪  DATA_WIDTH - 1 DOWNTO 0);
52    -- the Bram address of the ofms
53    SIGNAL addr_ofm : NATURAL RANGE 0 TO MAX_OFMS - 1;
54
55    -- provides the address to write to when new ofms are written
```

```
56    SIGNAL write_counter, write_counter_nxt : NATURAL RANGE 0 TO
      ↪  OFM_SIZE_IFMAP - 1;
57    -- when writing out only 8 bits can be written to the UART this is
      ↪  realized with these signals
58    SIGNAL uart_buffer, uart_buffer_nxt : STD_LOGIC_VECTOR(VAlUES_WIDTH
      ↪  * DATA_WIDTH - 1 DOWNTO 0);
59    SIGNAL data_counter, data_counter_nxt : NATURAL RANGE 0 TO
      ↪  PARALLEL_DATA - 1;
60    TYPE uart_state_type IS (IDLE, LOAD_DATA, STATIONARY, FINISHED);
61    -- represents the uart state as communicated from the UART
62    SIGNAL uart_state, uart_state_nxt : uart_state_type;
63    -- counter for keeping track when writing out over uart
64    -- (seperated from write_counter because it could be extended to
      ↪  overlap (saving some cycles), not implemented)
65    SIGNAL read_counter, read_counter_nxt : NATURAL RANGE 0 TO
      ↪  MAX_ADDR_DATA;
66    TYPE set_scale_state_type IS (IDLE, SET_SCALE, SET_OFFSET,
      ↪  WAIT_STATE);
67    SIGNAL set_scale_state, set_scale_state_nxt : set_scale_state_type;
68    SIGNAL addr_scale_counter, addr_scale_counter_nxt : NATURAL RANGE 0
      ↪  TO PARALLEL_OFMS + 7;
69
70    -- keeps track of where to write the ofms that come from the
      ↪  requant stage
71    SIGNAL iter_offs, iter_offs_nxt : NATURAL RANGE 0 TO MAX_OFMS - 1;
72    TYPE debug_buffer_type IS ARRAY(0 TO PARALLEL_OFMS - 1, 0 TO
      ↪  PE_COLUMNS - 1) OF STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
73
74    SIGNAL raddr_int : NATURAL RANGE 0 TO MAX_ADDR_DATA - 1;
75    CONSTANT OFFSET_DELAY : NATURAL := 3;
76    SIGNAL out_result_valid_nxt, out_result_valid : STD_LOGIC;
77
78    SIGNAL offset_counter, offset_counter_nxt : NATURAL RANGE 0 TO
      ↪  OFFSET_DELAY;
79    SIGNAL mult_data_nxt, mult_data : STD_LOGIC_VECTOR(PARALLEL_OFMS *
      ↪  PE_COLUMNS * ACC_DATA_WIDTH - 1 DOWNTO 0);
80    SIGNAL mult_data_valid_nxt, mult_data_valid : STD_LOGIC;
81    SIGNAL wait_counter, wait_counter_nxt : NATURAL RANGE 0 TO 100;
82
83    -- debug signals
84    TYPE debug_type IS ARRAY(0 TO PARALLEL_OFMS - 1, 0 TO PE_COLUMNS -
      ↪  1)OF unsigned(DATA_WIDTH - 1 DOWNTO 0);
85    TYPE debug_type_2 IS ARRAY(0 TO PARALLEL_OFMS - 1, 0 TO PE_COLUMNS
      ↪  - 1)OF signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
86    SIGNAL debug : debug_type;
87    SIGNAL debug_2 : debug_type_2;
88    SIGNAL debug_uart_buffer : debug_buffer_type;
89
90    -- used for the requantization
```

```
91    -- requant pipeline signals
92    TYPE result_mult_type IS ARRAY(0 TO PARALLEL_OFMS - 1, 0 TO
      ↪  PE_COLUMNS - 1) OF signed(ACC_DATA_WIDTH + 32 - 1 DOWNTO 0);
93    TYPE result_type IS ARRAY(0 TO PARALLEL_OFMS - 1, 0 TO PE_COLUMNS -
      ↪  1) OF signed(DATA_WIDTH - 1 + 1 DOWNTO 0);
94    SIGNAL result_mult_nxt, result_mult, out_result_reg,
      ↪  out_result_reg_nxt : result_mult_type;
95    SIGNAL result, result_nxt : result_type;
96    SIGNAL result_valid, result_valid_nxt : STD_LOGIC;
97    SIGNAL result_mult_valid, result_mult_valid_nxt : STD_LOGIC;
98
99    -- control when to advance the 'scales' (requant) i.e. the values
      ↪  that are used to requantize the 24 ACC-WIDTH BITS to 8bits
100   SIGNAL counter, counter_nxt : NATURAL RANGE 0 TO MAX_ADDR_ITER - 1;
101   -- the 'shift' value for requantization
102   SIGNAL dout_shift : STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
103   TYPE scale_type IS ARRAY(0 TO PARALLEL_OFMS - 1) OF signed(32 - 1
      ↪  DOWNTO 0);
104   TYPE shift_type IS ARRAY(0 TO PARALLEL_OFMS - 1) OF NATURAL RANGE 0
      ↪  TO 255 - 1;
105   SIGNAL scale_nxt, scale : scale_type;
106   SIGNAL shift, shift_nxt : shift_type;
107   SIGNAL scale_buffer, scale_buffer_nxt : scale_type;
108   SIGNAL shift_buffer, shift_buffer_nxt : shift_type;
109
110   -- it is important that we round away from zero see thesis
111   FUNCTION round_away_from_zero(vec : signed(ACC_DATA_WIDTH + 32 - 1
      ↪  DOWNTO 0);
112     n : NATURAL RANGE 0 TO 255 - 1) RETURN signed IS
113     VARIABLE result : signed(9 - 1 DOWNTO 0) := (OTHERS => '0');
114     VARIABLE neg : STD_LOGIC; --tells if negative
115
116   BEGIN
117     IF n < 2 THEN
118       RETURN to_signed(0, DATA_WIDTH + 1);
119     END IF;
120     result := shift_right(vec, n)(9 - 1 DOWNTO 0);
121     neg := STD_LOGIC(vec(vec'length - 1));
122     IF neg = '1' THEN
123       IF vec(n - 1) = '1' THEN
124         RETURN result + 1;
125       ELSE --first place is '1'
126         RETURN result;
127       END IF;
128     ELSE -- positve
129       IF vec(n - 1) = '0' THEN
130         RETURN result;
131       ELSE --first place is a '1'
132         RETURN result + 1;
```

```
133         END IF;
134       END IF;
135       RETURN result;
136     END FUNCTION;
137   BEGIN
138
139     sync : PROCESS (clk, reset)
140     BEGIN
141       IF rising_edge(clk) THEN
142         IF reset = '0' THEN
143           shift <= (OTHERS => 0);
144           scale <= (OTHERS => (OTHERS => '0'));
145           counter <= 0;
146           result_mult <= (OTHERS => (OTHERS => to_signed(0,
              ↪  result_mult(0, 0)'length)));
147           result <= (OTHERS => (OTHERS => to_signed(0, DATA_WIDTH +
              ↪  1)));
148           result_valid <= '0';
149           result_mult_valid <= '0';
150           read_counter <= 0;
151           data_counter <= 0;
152           uart_state <= IDLE;
153           uart_buffer <= (OTHERS => '0');
154           write_counter <= 0;
155           addr_scale_counter <= 0;
156           set_scale_state <= SET_SCALE;
157           scale_buffer <= (OTHERS => (OTHERS => '0'));
158           shift_buffer <= (OTHERS => 0);
159           iter_offs <= 0;
160           offset_counter <= 0;
161           out_result_valid <= '0';
162           out_result_reg <= (OTHERS => (OTHERS => to_signed(0,
              ↪  result_mult(0, 0)'length)));
163           mult_data <= (OTHERS => '0');
164           mult_data_valid <= '0';
165           wait_counter <= 0;
166         ELSE
167           set_scale_state <= set_scale_state_nxt;
168           iter_offs <= iter_offs_nxt;
169           addr_scale_counter <= addr_scale_counter_nxt;
170           scale_buffer <= scale_buffer_nxt;
171           shift <= shift_nxt;
172           scale <= scale_nxt;
173           counter <= counter_nxt;
174           result_mult <= result_mult_nxt;
175           result <= result_nxt;
176           result_valid <= result_valid_nxt;
177           result_mult_valid <= result_mult_valid_nxt;
178           read_counter <= read_counter_nxt;
```

```
179           uart_state <= uart_state_nxt;
180           data_counter <= data_counter_nxt;
181           uart_buffer <= uart_buffer_nxt;
182           write_counter <= write_counter_nxt;
183           shift_buffer <= shift_buffer_nxt;
184           offset_counter <= offset_counter_nxt;
185           out_result_valid <= out_result_valid_nxt;
186           out_result_reg <= out_result_reg_nxt;
187           mult_data <= mult_data_nxt;
188           mult_data_valid <= mult_data_valid_nxt;
189           wait_counter <= wait_counter_nxt;
190         END IF;
191       END IF;
192
193     END PROCESS;
194
195     --requants
196     requant_pipeline : PROCESS (ALL)
197       VARIABLE out_var : unsigned(DATA_WIDTH - 1 DOWNTO 0);
198     BEGIN
199       we <= '0';
200       waddr <= (OTHERS => '0');
201       ena <= '0';
202       result_valid_nxt <= '0';
203       din <= (OTHERS => '0');
204       write_counter_nxt <= write_counter;
205       result_mult_valid_nxt <= '0';
206       result_nxt <= result;
207       result_mult_nxt <= result_mult;
208       out_result_valid_nxt <= '0';
209       mult_data_nxt <= ofms_in.data;
210       mult_data_valid_nxt <= ofms_in.valid;
211
212       IF mult_data_valid = '1' THEN
213         FOR ofm IN 0 TO PARALLEL_OFMS - 1 LOOP
214           FOR I IN 0 TO PE_COLUMNS - 1 LOOP
215             result_mult_nxt(ofm, I) <= signed(mult_data(ACC_DATA_WIDTH
              ↪  * (I + 1 + ofm * PE_COLUMNS) - 1 DOWNTO ACC_DATA_WIDTH
              ↪  * (I + ofm * PE_COLUMNS))) * scale(ofm);
216             debug_2(ofm,I) <=
              ↪  signed(mult_data(ACC_DATA_WIDTH*(I+1+ofm*PE_COLUMNS)-1
              ↪  downto ACC_DATA_WIDTH*(I+ofm*PE_COLUMNS)));
217           END LOOP;
218         END LOOP;
219         result_mult_valid_nxt <= '1';
220       END IF;
221
222       -- this is added for better DSP inference (extra register stage
      ↪  -> power savings)
```

```
223       -- this doesn't work if compiled with to relaxed constraints (see
          ↪  Xilinx Forums)
224       out_result_reg_nxt <= result_mult;
225       IF result_mult_valid = '1' THEN
226         out_result_valid_nxt <= '1';
227       END IF;
228
229       IF out_result_valid = '1' THEN
230         result_valid_nxt <= '1';
231         FOR ofm IN 0 TO PARALLEL_OFMS - 1 LOOP
232           FOR I IN 0 TO PE_COLUMNS - 1 LOOP
233             result_nxt(ofm, I) <=
                ↪  round_away_from_zero(out_result_reg(ofm, I),
                ↪  shift(ofm));
234             --  debug(I) <= shift_right(result_mult(I),shift);
235           END LOOP;
236         END LOOP;
237       END IF;
238       waddr <= (OTHERS => '0');
239
240       IF result_valid = '1' THEN
241         we <= '1';
242         ena <= '1';
243         IF write_counter = OFM_SIZE_IFMAP - 1 THEN
244           write_counter_nxt <= 0;
245         ELSE
246           write_counter_nxt <= write_counter + 1;
247         END IF;
248         waddr <= STD_LOGIC_VECTOR(to_unsigned(write_counter + iter_offs
          ↪  * OFM_SIZE_IFMAP, waddr'length));
249         FOR ofm IN 0 TO PARALLEL_OFMS - 1 LOOP
250           FOR I IN 0 TO PE_COLUMNS - 1 LOOP
251
252             out_var := to_unsigned(to_integer(result(ofm, I)) +
                ↪  ofm_quant, DATA_WIDTH);
253             din(DATA_WIDTH * (I + 1 + ofm * PE_COLUMNS) - 1 DOWNTO
                ↪  DATA_WIDTH * (I + ofm * PE_COLUMNS)) <=
                ↪  STD_LOGIC_VECTOR(out_var);
254             --out_var :=
                ↪  to_unsigned(to_integer(result(I)+ofm_quant),DATA_WIDTH);
255             debug(ofm,I) <= out_var;
256           END LOOP;
257         END LOOP;
258       END IF;
259       CASE(uart_state) IS
260
261         WHEN IDLE =>
262         WHEN OTHERS =>
263         ena <= '1';
```

```
264       END CASE;
265     END PROCESS;
266
267     --prepares the needed shift and scale values
268     addr : PROCESS (ALL)
269     BEGIN
270       counter_nxt <= counter;
271       IF ofms_in.valid = '1' THEN
272         counter_nxt <= counter + 1;
273       END IF;
274       IF counter = OFM_SIZE_IFMAP - 1 THEN
275         counter_nxt <= 0;
276       END IF;
277       set_scale_state_nxt <= set_scale_state;
278       addr_scale_counter_nxt <= 0;
279       addr_ofm <= 0;
280       scale_nxt <= scale;
281       shift_nxt <= shift;
282       iter_offs_nxt <= iter_offs;
283       offset_counter_nxt <= 0;
284       shift_nxt <= shift;
285       scale_buffer_nxt <= scale_buffer;
286       shift_buffer_nxt <= shift_buffer;
287       wait_counter_nxt <= 0;
288       CASE(set_scale_state) IS
289
290         WHEN IDLE =>
291         IF counter = OFM_SIZE_IFMAP - 1 THEN
292           set_scale_state_nxt <= WAIT_STATE;
293         END IF;
294         WHEN WAIT_STATE =>
295         IF wait_counter = 100 THEN
296           set_scale_state_nxt <= SET_OFFSET;
297         ELSE
298           wait_counter_nxt <= wait_counter + 1;
299         END IF;
300
301         WHEN SET_OFFSET =>
302         IF offset_counter = OFFSET_DELAY - 1 THEN
303           set_scale_state_nxt <= SET_SCALE;
304           IF (iter_offs + 1) * PARALLEL_OFMS = MAX_OFMS THEN
305             iter_offs_nxt <= 0;
306           ELSE
307             iter_offs_nxt <= iter_offs + 1;
308           END IF;
309         ELSE
310           offset_counter_nxt <= offset_counter + 1;
311         END IF;
312
```

```
313         WHEN SET_SCALE =>
314         IF addr_scale_counter = PARALLEL_OFMS + 4 THEN
315           set_scale_state_nxt <= IDLE;
316           scale_nxt <= scale_buffer;
317           shift_nxt <= shift_buffer;
318         ELSE
319           addr_scale_counter_nxt <= addr_scale_counter + 1;
320         END IF;
321         IF addr_scale_counter > 0 AND addr_scale_counter <
          ↪  PARALLEL_OFMS + 1 THEN
322           scale_buffer_nxt(addr_scale_counter - 1) <=
            ↪  signed(dout_scale);
323           shift_buffer_nxt(addr_scale_counter - 1) <=
            ↪  to_integer(unsigned(dout_shift));
324         END IF;
325         IF addr_scale_counter < PARALLEL_OFMS THEN
326           addr_ofm <= addr_scale_counter + iter_offs * PARALLEL_OFMS;
327         END IF;
328       END CASE;
329     END PROCESS;
330
331     -- process for communicating with the UART unit
332     write_out_over_UART : PROCESS (ALL)
333       VARIABLE uart_buffer_var : STD_LOGIC_VECTOR(VAlUES_WIDTH *
        ↪  DATA_WIDTH - 1 DOWNTO 0);
334     BEGIN
335       read_counter_nxt <= 0;
336       to_uart.valid <= '0';
337       to_uart.data <= (OTHERS => '0');
338       uart_state_nxt <= uart_state;
339       data_counter_nxt <= data_counter;
340       uart_buffer_nxt <= uart_buffer;
341       IF read_counter < MAX_ADDR_DATA THEN
342         raddr_int <= read_counter;
343       ELSE
344         raddr_int <= 0;
345       END IF;
346
347       CASE(uart_state) IS
348
349         WHEN IDLE =>
350         to_uart.valid <= '0';
351         IF from_uart.want_data_ofm = '1' THEN
352           uart_state_nxt <= LOAD_DATA;
353
354         ELSE
355           uart_state_nxt <= IDLE;
356         END IF;
357
```

```
358         WHEN LOAD_DATA =>
359         read_counter_nxt <= read_counter;
360         uart_buffer_var := dout;
361         data_counter_nxt <= 0;
362         IF from_uart.ready = '1' THEN
363           IF 0 = PARALLEL_DATA - 1 THEN
364             data_counter_nxt <= 0;
365           ELSE
366             data_counter_nxt <= 1;
367           END IF;
368           to_uart.data <= uart_buffer_var(DATA_WIDTH - 1 DOWNTO 0);
369
370           to_uart.valid <= '1';
371
372           uart_buffer_nxt <= uart_buffer_var;
373           FOR ofm IN 0 TO PARALLEL_OFMS - 1 LOOP
374             FOR I IN 0 TO PE_COLUMNS - 1 LOOP
375               debug_uart_buffer(ofm, I) <= uart_buffer_var(DATA_WIDTH *
                  ↪  (I + ofm * PE_COLUMNS + 1) - 1 DOWNTO DATA_WIDTH * (I
                  ↪  + ofm * PE_COLUMNS));
376             END LOOP;
377           END LOOP;
378           IF 0 = PARALLEL_DATA - 1 THEN
379             uart_state_nxt <= LOAD_DATA;
380           ELSE
381             uart_state_nxt <= STATIONARY;
382           END IF;
383
384           IF read_counter = MAX_ADDR_DATA THEN
385             read_counter_nxt <= 0;
386             uart_state_nxt <= FINISHED;
387           ELSE
388             read_counter_nxt <= read_counter + 1;
389           END IF;
390         END IF;
391         WHEN FINISHED =>
392         to_uart.valid <= '0';
393
394         WHEN STATIONARY =>
395         read_counter_nxt <= read_counter;
396         IF from_uart.ready = '1' THEN
397           to_uart.data <= uart_buffer(DATA_WIDTH * (data_counter + 1) -
            ↪  1 DOWNTO DATA_WIDTH * data_counter);
398           to_uart.valid <= '1';
399           IF data_counter = PARALLEL_DATA - 1 THEN
400             uart_state_nxt <= LOAD_DATA;
401           ELSE
402             data_counter_nxt <= data_counter + 1;
403           END IF;
```

```
404         END IF;
405       END CASE;
406
407     END PROCESS;
408
409     --scale memory
410     scales_mem : ENTITY work.rams_init_file
411       GENERIC MAP(
412         FILENAME => "scales.data",
413         ADDRW => 10, --255*3
414         DATAW => 32,
415         DEPTH => MAX_OFMS * MAX_RATE
416       )
417       PORT MAP(
418         clk => clk,
419         addr => STD_LOGIC_VECTOR(to_unsigned(addr_ofm, 10)),
420         dout => dout_scale
421       );
422
423     shift_mem : ENTITY work.rams_init_file
424       GENERIC MAP(
425         FILENAME => "shift.data",
426         ADDRW => 10, --255*3
427         DATAW => DATA_WIDTH,
428         DEPTH => MAX_OFMS * MAX_RATE
429       )
430       PORT MAP(
431         clk => clk,
432         addr => STD_LOGIC_VECTOR(to_unsigned(addr_ofm, 10)),
433         dout => dout_shift
434       );
435     --ofm memory
436     rams_sdp_record_i : ENTITY work.rams_sdp_record
437       GENERIC MAP(
438         A_WID => A_WID,
439         D_WID => PARALLEL_OFMS * PE_COLUMNS * DATA_WIDTH,
440         DEPTH => MAX_ADDR_DATA
441       )
442       PORT MAP(
443         clk => clk,
444         we => we,
445         ena => ena,
446         raddr => STD_LOGIC_VECTOR(to_unsigned(raddr_int, A_WID)),
447         waddr => waddr,
448         din => din,
449         dout => dout
450       );
451   END ARCHITECTURE;
```

PE/accum.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4  USE work.core_pck.ALL;
5
6  ENTITY accum_unit IS
7    GENERIC (
8      ACC_DATA_WIDTH : NATURAL := 24;
9      DATA_WIDTH_RESULT : NATURAL := 18
10   );
11   PORT (
12
13     reset, clk : IN STD_LOGIC;
14     new_psum : IN STD_LOGIC;
15     new_ifmap : IN STD_LOGIC;
16     finished_in : IN STD_LOGIC;
17     finished_out : OUT STD_LOGIC;
18     psum_in : IN signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
19     psum_out : OUT signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
20     result : IN signed(DATA_WIDTH_RESULT - 1 DOWNTO 0);
21     valid : IN STD_LOGIC
22   );
23 END ENTITY;
24
25 ARCHITECTURE arch OF accum_unit IS
26   SIGNAL finished_reg, finished, finished_nxt : STD_LOGIC;
27   TYPE psum_array IS ARRAY (0 TO 1) OF signed(ACC_DATA_WIDTH - 1
     ↪  DOWNTO 0);
28   SIGNAL psum, psum_nxt : psum_array;
29   SIGNAL swap, swap_nxt : STD_LOGIC_VECTOR(0 DOWNTO 0);
30 BEGIN
31   sync : PROCESS (clk, reset)
32   BEGIN
33     IF reset = '0' THEN
34       psum <= (OTHERS => (OTHERS => '0'));
35       swap <= "0";
36       finished <= '0';
37     ELSIF rising_edge(clk) THEN
38       psum <= psum_nxt;
39       finished_reg <= finished;
40       swap <= swap_nxt;
41       finished <= finished_nxt;
42     END IF;
43   END PROCESS;
44
45   -- is double buffered and while accumulating one psum the other one
     ↪  is read and swapped out
```

```
46    state : PROCESS (ALL)
47      VARIABLE swap_int, not_swap_int : NATURAL;
48    BEGIN
49      swap_int := to_integer(unsigned(swap));
50      not_swap_int := to_integer(unsigned(NOT(swap)));
51      psum_nxt <= psum;
52      swap_nxt <= swap;
53      IF new_psum = '1' THEN
54        psum_nxt(not_swap_int) <= psum_in;
55      END IF;
56
57      IF new_ifmap = '1' THEN
58        swap_nxt <= NOT(swap);
59      END IF;
60      IF valid = '1' THEN
61        psum_nxt(swap_int) <= psum(swap_int) + resize(result,
          ↪  ACC_DATA_WIDTH);
62      END IF;
63      finished_nxt <= finished_in;
64      finished_out <= finished;
65      psum_out <= psum(not_swap_int);
66
67    END PROCESS;
68
69  END ARCHITECTURE;
```

PE/fetch_unit.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3
4   USE ieee.numeric_std.ALL;
5   USE work.core_pck.ALL;
6
7   USE work.fetch_unit_pck.ALL;
8
9   -- The fetch unit is responsible for extracting the next valid index
10  -- (neither weight and ifmap equal to zero)
11  ENTITY fetch_unit IS
12    GENERIC (
13      COMPARISON_BITVEC_WIDTH : NATURAL := 18
14    );
15    PORT (
16      reset, clk : IN STD_LOGIC;
17      finished : OUT STD_LOGIC;
18      new_kernels : IN STD_LOGIC;
19      new_ifmaps : IN STD_LOGIC;
20      kernel_bitvecs, ifmap_bitvecs : IN STD_LOGIC_VECTOR(FILTER_PER_PE
        ↪  - 1 DOWNTO 0);
```

```
21      index : OUT NATURAL RANGE 0 TO FILTER_PER_PE - 1;
22      valid : OUT STD_LOGIC
23    );
24  END fetch_unit;
25
26  ARCHITECTURE arch OF fetch_unit IS
27    SIGNAL state, state_nxt : state_type;
28    SIGNAL kernel_bitvecs_reg, kernel_bitvecs_reg_storage,
      ↪  kernel_bitvecs_reg_storage_nxt, kernel_bitvecs_reg_nxt :
      ↪  STD_LOGIC_VECTOR(FILTER_PER_PE - 1 DOWNTO 0);
29    SIGNAL ifmap_bitvecs_reg, ifmap_bitvecs_reg_nxt :
      ↪  STD_LOGIC_VECTOR(FILTER_PER_PE - 1 DOWNTO 0);
30    CONSTANT MAX_COUNTER : NATURAL :=
      ↪  FILTER_PER_PE/COMPARISON_BITVEC_WIDTH;
31    SIGNAL index_reg, index_reg_nxt : NATURAL RANGE 0 TO FILTER_PER_PE
      ↪  + COMPARISON_BITVEC_WIDTH;
32
33  BEGIN
34
35    sync : PROCESS (clk, reset)
36    BEGIN
37      IF reset = '0' THEN
38        state <= LOADING_VALUES;
39        ifmap_bitvecs_reg <= (OTHERS => '0');
40        kernel_bitvecs_reg <= (OTHERS => '0');
41        index_reg <= 0;
42        kernel_bitvecs_reg_storage <= (OTHERS => '0');
43      ELSIF rising_edge(clk) THEN
44        state <= state_nxt;
45        kernel_bitvecs_reg <= kernel_bitvecs_reg_nxt;
46        ifmap_bitvecs_reg <= ifmap_bitvecs_reg_nxt;
47        index_reg <= index_reg_nxt;
48        kernel_bitvecs_reg_storage <= kernel_bitvecs_reg_storage_nxt;
49      END IF;
50    END PROCESS;
51
52    -- see thesis for documentation
53    state_process : PROCESS (ALL)
54      VARIABLE comp_window : STD_LOGIC_VECTOR(COMPARISON_BITVEC_WIDTH -
        ↪  1 DOWNTO 0);
55      VARIABLE valid_var : STD_LOGIC;
56      VARIABLE index_var : NATURAL RANGE 0 TO COMPARISON_BITVEC_WIDTH -
        ↪  1;
57      VARIABLE index_comp : NATURAL RANGE 0 TO FILTER_PER_PE +
        ↪  COMPARISON_BITVEC_WIDTH - 1;
58    BEGIN
59      valid_var := '0';
60      state_nxt <= state;
61      valid <= '0';
```

```
62      finished <= '0';
63      ifmap_bitvecs_reg_nxt <= ifmap_bitvecs_reg;
64      kernel_bitvecs_reg_storage_nxt <= kernel_bitvecs_reg_storage;
65      kernel_bitvecs_reg_nxt <= kernel_bitvecs_reg;
66      index <= 0;
67      index_reg_nxt <= 0;
68      CASE(state) IS
69
70        WHEN LOADING_VALUES =>
71        kernel_bitvecs_reg_nxt <= kernel_bitvecs_reg_storage;
72        IF new_kernels = '1' THEN
73          kernel_bitvecs_reg_nxt <= kernel_bitvecs;
74          kernel_bitvecs_reg_storage_nxt <= kernel_bitvecs;
75        ELSIF new_ifmaps = '1' THEN
76          ifmap_bitvecs_reg_nxt <= ifmap_bitvecs;
77          state_nxt <= PROCESSING;
78        END IF;
79        finished <= '1';
80
81        WHEN PROCESSING =>
82        comp_window := kernel_bitvecs_reg((COMPARISON_BITVEC_WIDTH) - 1
          ↪  DOWNTO 0) AND ifmap_bitvecs_reg((COMPARISON_BITVEC_WIDTH) -
          ↪  1 DOWNTO 0);
83        mask_last(comp_window, index_var, valid_var);
84        index_comp := index_var + index_reg;
85        kernel_bitvecs_reg_nxt <=
          ↪  STD_LOGIC_VECTOR(shift_right(unsigned(kernel_bitvecs_reg),
          ↪  index_var + 1));
86        ifmap_bitvecs_reg_nxt <=
          ↪  STD_LOGIC_VECTOR(shift_right(unsigned(ifmap_bitvecs_reg),
          ↪  index_var + 1));
87        IF index_comp < FILTER_PER_PE THEN
88          valid <= valid_var;
89        ELSE
90          state_nxt <= LOADING_VALUES;
91          valid <= '0';
92          index_comp := 0;
93          finished <= '1';
94        END IF;
95        index <= index_comp;
96        index_reg_nxt <= index_comp + 1;
97      END CASE;
98    END PROCESS;
99
100 END ARCHITECTURE;
```

PE/fetch_unit_pck.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4  USE work.core_pck.ALL;
5  USE work.pe_pack.ALL;
6
7  PACKAGE fetch_unit_pck IS
8    TYPE state_type IS (LOADING_VALUES, PROCESSING);
9
10   PROCEDURE mask_last (VARIABLE bitvec_var : INOUT
     ↪  STD_LOGIC_VECTOR(COMPARISON_BITVEC_WIDTH - 1 DOWNTO 0);
11   VARIABLE index_var : OUT NATURAL RANGE 0 TO COMPARISON_BITVEC_WIDTH
     ↪  - 1;
12   VARIABLE valid_var : OUT STD_LOGIC);
13 END PACKAGE;
14 PACKAGE BODY fetch_unit_pck IS
15
16   PROCEDURE mask_last (VARIABLE bitvec_var : INOUT
     ↪  STD_LOGIC_VECTOR(COMPARISON_BITVEC_WIDTH - 1 DOWNTO 0);
17   VARIABLE index_var : OUT NATURAL RANGE 0 TO COMPARISON_BITVEC_WIDTH
     ↪  - 1;
18   VARIABLE valid_var : OUT STD_LOGIC) IS
19 BEGIN
20   valid_var := '0';
21   index_var := COMPARISON_BITVEC_WIDTH - 1;
22   FOR I IN bitvec_var'low TO bitvec_var'high LOOP
23     IF bitvec_var(I) = '1' THEN
24       valid_var := '1';
25       index_var := I;
26       --bitvec_var(I) := '0';
27       EXIT;
28     END IF;
29   END LOOP;
30
31 END PROCEDURE;
32 END PACKAGE BODY;
```

PE/mult.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4  USE work.core_pck.ALL;
5  ENTITY mult_unit IS
6    GENERIC (
7      DATA_WIDTH_RESULT : NATURAL := 18
8    );
```

```
9     PORT (
10      clk : IN STD_LOGIC;
11      reset : IN STD_LOGIC;
12      finished_in : IN STD_LOGIC;
13      finished_out : OUT STD_LOGIC;
14      new_kernels : IN STD_LOGIC;
15      new_ifmaps : IN STD_LOGIC;
16      data : IN STD_LOGIC_VECTOR(DATA_WIDTH * FILTER_PER_PE - 1 DOWNTO
        ↪  0);
17      ifmap_zero_offset : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
18      index : IN NATURAL RANGE 0 TO FILTER_PER_PE - 1;
19      valid : IN STD_LOGIC;
20      valid_out : OUT STD_LOGIC;
21      result_out : OUT signed(DATA_WIDTH_RESULT - 1 DOWNTO 0)
22    );
23  END ENTITY;
24
25  ARCHITECTURE arch OF mult_unit IS
26    TYPE kernel_reg_type IS ARRAY (0 TO FILTER_PER_PE - 1) OF
      ↪  signed(DATA_WIDTH - 1 DOWNTO 0);
27    TYPE ifmap_reg_type IS ARRAY (0 TO FILTER_PER_PE - 1) OF
      ↪  unsigned(DATA_WIDTH - 1 DOWNTO 0);
28    SIGNAL kernel_value_reg, kernel_value_reg_nxt : kernel_reg_type;
29    SIGNAL ifmap_value_reg, ifmap_value_reg_nxt : ifmap_reg_type;
30    SIGNAL ifmap_reg_signed, ifmap_reg_signed_nxt : signed(DATA_WIDTH -
      ↪  1 + 1 DOWNTO 0);
31    SIGNAL ifmap_zero_reg_nxt, ifmap_zero_reg : unsigned(DATA_WIDTH - 1
      ↪  DOWNTO 0);
32    SIGNAL valid_mult, valid_mult_nxt, valid_result, valid_result_nxt,
      ↪  finished_result, finished_result_nxt, finished_prep_nxt,
      ↪  finished_prep : STD_LOGIC;
33    SIGNAL result, result_nxt : signed(DATA_WIDTH_RESULT - 1 DOWNTO 0);
34    SIGNAL weight_reg_signed_nxt, weight_reg_signed : signed(DATA_WIDTH
      ↪  - 1 + 1 DOWNTO 0);
35  BEGIN
36    sync : PROCESS (clk, reset)
37    BEGIN
38      IF reset = '0' THEN
39        kernel_value_reg <= (OTHERS => (OTHERS => '0'));
40        ifmap_value_reg <= (OTHERS => (OTHERS => '0'));
41        ifmap_reg_signed <= (OTHERS => '0');
42        ifmap_zero_reg <= (OTHERS => '0');
43        valid_mult <= '0';
44        valid_result <= '0';
45        weight_reg_signed <= (OTHERS => '0');
46        result <= (OTHERS => '0');
47        finished_prep <= '0';
48        finished_result <= '0';
49
```

```
50      ELSIF rising_edge(clk) THEN
51        kernel_value_reg <= kernel_value_reg_nxt;
52        ifmap_value_reg <= ifmap_value_reg_nxt;
53        ifmap_reg_signed <= ifmap_reg_signed_nxt;
54        ifmap_zero_reg <= ifmap_zero_reg_nxt;
55        valid_mult <= valid_mult_nxt;
56        valid_result <= valid_result_nxt;
57        weight_reg_signed <= weight_reg_signed_nxt;
58        result <= result_nxt;
59        finished_prep <= finished_prep_nxt;
60        finished_result <= finished_result_nxt;
61      END IF;
62    END PROCESS;
63
64    -- fetches and stores
65    new_data : PROCESS (ALL)
66    BEGIN
67      kernel_value_reg_nxt <= kernel_value_reg;
68      ifmap_value_reg_nxt <= ifmap_value_reg;
69      ifmap_zero_reg_nxt <= unsigned(ifmap_zero_offset);
70
71      IF new_kernels = '1' THEN
72        FOR I IN 0 TO FILTER_PER_PE - 1 LOOP
73          kernel_value_reg_nxt(I) <= signed(data(DATA_WIDTH * (I + 1) -
            ↪  1 DOWNTO DATA_WIDTH * I));
74        END LOOP;
75      ELSIF new_ifmaps = '1' THEN
76        FOR I IN 0 TO FILTER_PER_PE - 1 LOOP
77          ifmap_value_reg_nxt(I) <= unsigned(data(DATA_WIDTH * (I + 1)
            ↪  - 1 DOWNTO DATA_WIDTH * I));
78        END LOOP;
79      END IF;
80    END PROCESS;
81
82    --multiplication pipeline
83    mult : PROCESS (ALL)
84    BEGIN
85      valid_out <= '0';
86      valid_mult_nxt <= valid;
87      result_nxt <= (OTHERS => '0');
88      ifmap_reg_signed_nxt <= (OTHERS => '0');
89      weight_reg_signed_nxt <= (OTHERS => '0');
90      valid_result_nxt <= '0';
91      finished_prep_nxt <= finished_in;
92      finished_result_nxt <= finished_prep;
93      finished_out <= finished_result;
94      IF valid = '1' THEN
```

```
95        ifmap_reg_signed_nxt <=
          ↪  to_signed(to_integer(ifmap_value_reg(index)) -
          ↪  to_integer(ifmap_zero_reg), ifmap_reg_signed'length);
96        weight_reg_signed_nxt <= resize(kernel_value_reg(index),
          ↪  weight_reg_signed'length);
97      END IF;
98      IF valid_mult = '1' THEN
99        result_nxt <= ifmap_reg_signed * weight_reg_signed;
100       valid_result_nxt <= valid_mult;
101     END IF;
102     valid_out <= valid_result;
103     result_out <= result;
104   END PROCESS;
105
106 END ARCHITECTURE;
```

PE/pe.vhd

```
1   -- The PE
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.ALL;
4   USE ieee.numeric_std.ALL;
5   USE work.core_pck.ALL;
6   USE work.pe_pack.ALL;
7
8   ENTITY pe IS
9     PORT (
10      reset, clk : IN STD_LOGIC;
11      new_kernels : IN STD_LOGIC;
12      new_ifmaps : IN STD_LOGIC;
13      new_psum : IN STD_LOGIC;
14      psum_in : IN signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
15      bus_to_pe : IN STD_LOGIC_VECTOR(BUSSIZE - 1 DOWNTO 0);
16      psum : OUT signed(ACC_DATA_WIDTH - 1 DOWNTO 0);
17      mult_counter : OUT unsigned(EXEC_COUNTER_WIDTH - 1 DOWNTO 0);
18      ifmap_zero_offset : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
19      finished_ifmaps : OUT STD_LOGIC
20    );
21  END ENTITY;
22
23  ARCHITECTURE arch OF pe IS
24
25    SIGNAL valid_from_fetch, valid_from_mult, finished_from_mult,
      ↪  finished_from_fetch, finished : STD_LOGIC;
26    SIGNAL index : NATURAL RANGE 0 TO FILTER_PER_PE - 1;
27    SIGNAL result : signed(DATA_WIDTH_RESULT - 1 DOWNTO 0);
28    ALIAS bitvecs : STD_LOGIC_VECTOR(FILTER_PER_PE - 1 DOWNTO 0) IS
      ↪  bus_to_pe(FILTER_PER_PE - 1 DOWNTO 0);
```

```
29    ALIAS data : STD_LOGIC_VECTOR(FILTER_PER_PE * DATA_WIDTH - 1 DOWNTO
      ↪  0) IS bus_to_pe(FILTER_PER_PE * DATA_WIDTH - 1 + FILTER_PER_PE
      ↪  DOWNTO FILTER_PER_PE);
30
31  BEGIN
32    f_as : PROCESS (ALL)
33    BEGIN
34      finished_ifmaps <= finished_from_fetch;
35    END PROCESS;
36
37    fetch_unit_i : ENTITY work.fetch_unit
38      GENERIC MAP(
39        COMPARISON_BITVEC_WIDTH => COMPARISON_BITVEC_WIDTH
40      )
41      PORT MAP(
42        reset => reset,
43        clk => clk,
44        finished => finished_from_fetch,
45        new_kernels => new_kernels,
46        new_ifmaps => new_ifmaps,
47        kernel_bitvecs => bitvecs,
48        ifmap_bitvecs => bitvecs,
49        index => index,
50        valid => valid_from_fetch
51      );
52
53    mult_unit_i : ENTITY work.mult_unit
54      GENERIC MAP(
55        DATA_WIDTH_RESULT => DATA_WIDTH_RESULT
56      )
57      PORT MAP(
58        clk => clk,
59        reset => reset,
60        finished_in => finished_from_fetch,
61        finished_out => finished_from_mult,
62        new_kernels => new_kernels,
63        new_ifmaps => new_ifmaps,
64        data => data,
65        ifmap_zero_offset => ifmap_zero_offset,
66        index => index,
67        valid => valid_from_fetch,
68        valid_out => valid_from_mult,
69        result_out => result
70      );
71
72    accum_unit_i : ENTITY work.accum_unit
73      GENERIC MAP(
74        ACC_DATA_WIDTH => ACC_DATA_WIDTH,
75        DATA_WIDTH_RESULT => DATA_WIDTH_RESULT
```

```
76      )
77      PORT MAP(
78        reset => reset,
79        clk => clk,
80        new_psum => new_psum,
81        new_ifmap => new_ifmaps,
82        finished_in => finished_from_mult,
83        finished_out => finished,
84        psum_in => psum_in,
85        psum_out => psum,
86        result => result,
87        valid => valid_from_mult
88      );
89
90    mult_cp : PROCESS (clk, reset)
91    BEGIN
92      IF reset = '0' THEN
93        mult_counter <= (OTHERS => '0');
94      ELSIF rising_edge(clk) THEN
95        mult_counter <= mult_counter;
96        IF valid_from_mult = '1' THEN
97          mult_counter <= mult_counter + 1;
98        END IF;
99      END IF;
100   END PROCESS;
101 END ARCHITECTURE;
```

peripherals/in_unit_bram.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3
4   USE ieee.numeric_std.ALL;
5   USE work.core_pck.ALL;
6   USE work.control_pck.ALL;
7   USE work.top_types_pck.ALL;
8   USE IEEE.math_real.ALL;
9
10  ENTITY in_unit IS
11    GENERIC (
12      PARALLEL_OFMS : NATURAL := 3;
13      MAX_OFMS : NATURAL := 255;
14      MAX_RATE : NATURAL := 3;
15      PE_COLUMNS : NATURAL := 3;
16      FILTER_DEPTH : NATURAL := 1
17    );
18    PORT (
19      clk : IN STD_LOGIC;
20      reset : IN STD_LOGIC;
```

```
21      in_unit_to_ctrl : OUT in_unit_to_ctrl_type;
22      ctrl_to_in: IN ctrl_to_in_type
23    );
24  END ENTITY;
25
26  ARCHITECTURE arch OF in_unit IS
27    CONSTANT DEPTH_IFMAP : NATURAL := INTEGER((real(FILTER_DEPTH *
      ↪  IFMAP_SIZE * IFMAP_SIZE * 72 * 8/(288 * 2))));
28    CONSTANT ADDRW_IFMAP : NATURAL :=
      ↪  INTEGER(ceil(log2(real(FILTER_DEPTH * IFMAP_SIZE * IFMAP_SIZE *
      ↪  72 * 8/(288 * 2)))));
29    CONSTANT DEPTH_KERNEL : NATURAL := MAX_OFMS * FILTER_DEPTH * 9;
30    CONSTANT ADDRW_KERNEL : NATURAL :=
      ↪  INTEGER(ceil(log2(real(DEPTH_KERNEL))));
31    --the state of the ifmap buffer
32    TYPE state_buffer_t IS (IDLE, STATIONARY, OUTS, FINISHED);
33    SIGNAL state_ifmap, state_ifmap_nxt : state_buffer_t;
34    --the state of the kernel buffer
35    SIGNAL state_kernel, state_kernel_nxt : state_buffer_t;
36    --current read addr of ifmap/kernel
37    SIGNAL addr_ifmap, addr_ifmap_nxt : NATURAL RANGE 0 TO DEPTH_IFMAP
      ↪  - 1;
38    SIGNAL addr_kernel, addr_kernel_nxt : NATURAL RANGE 0 TO
      ↪  DEPTH_KERNEL - 1;
39
40    SIGNAL kernel_ofm_counter, kernel_ofm_counter_nxt : NATURAL RANGE 0
      ↪  TO PARALLEL_OFMS - 1 + 1;
41    -- ifmap and kernel values
42    SIGNAL ifmap : STD_LOGIC_VECTOR(72 * 8 - 1 DOWNTO 0);
43    SIGNAL kernel : STD_LOGIC_VECTOR(512 - 1 DOWNTO 0);
44    -- before providing new kernel values wait at least 100 cycles
      ↪  (could also be implemented by simply detecting edge)
45    -- 100 is ecessive it need only be ~2, however after kernel update
      ↪  it takes at least >1000 cycles until new ones are needed
46    SIGNAL wait_counter, wait_counter_nxt : NATURAL RANGE 0 TO 100;
47    TYPE deb_t IS ARRAY(0 TO 36 - 1) OF STD_LOGIC_VECTOR(DATA_WIDTH - 1
      ↪  DOWNTO 0);
48    SIGNAL debug_ifmap : deb_t;
49  BEGIN
50
51    debug : PROCESS (ALL)
52    BEGIN
53      FOR I IN 0 TO 36 - 1 LOOP
54        debug_ifmap(I) <= ifmap(DATA_WIDTH * (36 - I) - 1 DOWNTO
          ↪  DATA_WIDTH * (36 - 1 - I));
55      END LOOP;
56
57    END PROCESS;
58    sync : PROCESS (clk, reset)
```

```
59    BEGIN
60      IF reset = '0' THEN
61        state_ifmap <= IDLE;
62        state_kernel <= IDLE;
63        addr_ifmap <= 0;
64        addr_kernel <= 0;
65        wait_counter <= 0;
66        kernel_ofm_counter <= 0;
67      ELSIF rising_edge(clk) THEN
68        state_ifmap <= state_ifmap_nxt;
69        state_kernel <= state_kernel_nxt;
70        addr_ifmap <= addr_ifmap_nxt;
71        addr_kernel <= addr_kernel_nxt;
72        kernel_ofm_counter <= kernel_ofm_counter_nxt;
73        wait_counter <= wait_counter_nxt;
74      END IF;
75
76    END PROCESS;
77
78    ifmap_out : PROCESS (ALL)
79    BEGIN
80      addr_ifmap_nxt <= 0;
81      in_unit_to_ctrl.ifmap_values.valid <= '0';
82      in_unit_to_ctrl.ifmap_values.data <= (OTHERS => '0');
83      state_ifmap_nxt <= state_ifmap;
84      in_unit_to_ctrl.ifmaps_loaded <= '0';
85      CASE(state_ifmap) IS
86
87        WHEN IDLE =>
88        IF ctrl_to_in.load_ifmaps = '1' THEN
89          state_ifmap_nxt <= STATIONARY;
90          addr_ifmap_nxt <= 1;
91        END IF;
92
93        WHEN STATIONARY =>
94        state_ifmap_nxt <= STATIONARY;
95        FOR I IN 0 TO 72 - 1 LOOP
96          in_unit_to_ctrl.ifmap_values.data((I + 1) * DATA_WIDTH - 1
            ↪  DOWNTO DATA_WIDTH * I) <= ifmap((72 - I) * DATA_WIDTH - 1
            ↪  DOWNTO DATA_WIDTH * (72 - 1 - I));
97        END LOOP;
98        in_unit_to_ctrl.ifmap_values.valid <= '1';
99        IF addr_ifmap = DEPTH_IFMAP - 1 THEN
100         addr_ifmap_nxt <= 0;
101         state_ifmap_nxt <= OUTS;
102       ELSE
103         addr_ifmap_nxt <= addr_ifmap + 1;
104       END IF;
105       WHEN OUTS =>
```

```
106         FOR I IN 0 TO 72 - 1 LOOP
107           in_unit_to_ctrl.ifmap_values.data((I + 1) * DATA_WIDTH - 1
              ↪  DOWNTO DATA_WIDTH * I) <= ifmap((72 - I) * DATA_WIDTH - 1
              ↪  DOWNTO DATA_WIDTH * (72 - 1 - I));
108         END LOOP;
109         in_unit_to_ctrl.ifmap_values.valid <= '1';
110         state_ifmap_nxt <= FINISHED;
111
112         WHEN FINISHED =>
113         in_unit_to_ctrl.ifmaps_loaded <= '1';
114         in_unit_to_ctrl.ifmap_values.valid <= '0';
115       END CASE;
116     END PROCESS;
117     kernel_out : PROCESS (ALL)
118     BEGIN
119       state_kernel_nxt <= state_kernel;
120       addr_kernel_nxt <= addr_kernel;
121       kernel_ofm_counter_nxt <= 0;
122       in_unit_to_ctrl.new_kernels <= (OTHERS => '0');
123       in_unit_to_ctrl.kernels_loaded <= '0';
124       wait_counter_nxt <= wait_counter;
125       in_unit_to_ctrl.kernel_values <= (OTHERS => (OTHERS => '-'));
126       CASE (state_kernel) IS
127         WHEN IDLE =>
128           IF ctrl_to_in.load_kernels = '1' THEN
129             IF PARALLEL_OFMS = 1 THEN
130               state_kernel_nxt <= OUTS;
131             ELSE
132               state_kernel_nxt <= STATIONARY;
133               IF addr_kernel = DEPTH_KERNEL - 1 THEN
134                 addr_kernel_nxt <= 0;
135                 kernel_ofm_counter_nxt <= 0;
136               ELSE
137                 addr_kernel_nxt <= addr_kernel + 1;
138                 kernel_ofm_counter_nxt <= 0;
139               END IF;
140             END IF;
141           END IF;
142         WHEN STATIONARY =>
143           in_unit_to_ctrl.new_kernels(kernel_ofm_counter) <= '1';
144           kernel_ofm_counter_nxt <= kernel_ofm_counter + 1;
145           FOR I IN 0 TO 64 - 1 LOOP
146             in_unit_to_ctrl.kernel_values(I) <=
                ↪  signed(kernel(DATA_WIDTH * (64 - I) - 1 DOWNTO
                ↪  DATA_WIDTH * (63 - I)));
147           END LOOP;
148           IF kernel_ofm_counter = PARALLEL_OFMS - 2 THEN
149             state_kernel_nxt <= OUTS;
150           ELSE
```

```
151             IF addr_kernel = DEPTH_KERNEL - 1 THEN
152               addr_kernel_nxt <= 0;
153             ELSE
154               addr_kernel_nxt <= addr_kernel + 1;
155
156             END IF;
157           END IF;
158         WHEN OUTS =>
159           FOR I IN 0 TO 64 - 1 LOOP
160             in_unit_to_ctrl.new_kernels(PARALLEL_OFMS - 1) <= '1';
161             in_unit_to_ctrl.kernel_values(I) <=
                ↪  signed(kernel(DATA_WIDTH * (64 - I) - 1 DOWNTO
                ↪  DATA_WIDTH * (63 - I)));
162           END LOOP;
163           state_kernel_nxt <= FINISHED;
164           IF addr_kernel = DEPTH_KERNEL - 1 THEN
165             addr_kernel_nxt <= 0;
166           ELSE
167             addr_kernel_nxt <= addr_kernel + 1;
168
169           END IF;
170
171         WHEN FINISHED =>
172           in_unit_to_ctrl.kernels_loaded <= '1';
173
174           IF wait_counter = 50 THEN
175             wait_counter_nxt <= 0;
176             state_kernel_nxt <= IDLE;
177           ELSE
178             wait_counter_nxt <= wait_counter + 1;
179           END IF;
180       END CASE;
181
182     END PROCESS;
183     ifmap_mem : ENTITY work.rams_init_file
184       GENERIC MAP(
185         FILENAME => "ifmaps_mem.data",
186         ADDRW => ADDRW_IFMAP,
187         DATAW => 72 * 8,
188         DEPTH => DEPTH_IFMAP
189       )
190       PORT MAP(
191         clk => clk,
192         addr => STD_LOGIC_VECTOR(to_unsigned(addr_ifmap, ADDRW_IFMAP)),
193         dout => ifmap
194       );
195     kernel_mem : ENTITY work.rams_init_file
196       GENERIC MAP(
197         FILENAME => "kernels_mem.data",
```

```
198         ADDRW => ADDRW_KERNEL,
199         DATAW => 64 * 8,
200         DEPTH => DEPTH_KERNEL
201       )
202       PORT MAP(
203         clk => clk,
204         addr => STD_LOGIC_VECTOR(to_unsigned(addr_kernel,
            ↪  ADDRW_KERNEL)),
205         dout => kernel
206       );
207   END ARCHITECTURE;
```

peripherals/uart.vhd

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4   USE work.core_pck.ALL;
5   USE work.control_pck.ALL;
6   USE work.top_types_pck.ALL;
7
8   -- The uart writes out to the PC
9   ENTITY uart_unit IS
10    PORT (
11      clk : IN STD_LOGIC;
12      reset : IN STD_LOGIC;
13      from_uart : OUT from_uart_type;
14      to_uart : IN to_uart_type;
15      rx : IN STD_LOGIC;
16      tx : OUT STD_LOGIC;
17      finished : IN STD_LOGIC;
18      finished_counters : IN STD_LOGIC
19    );
20  END ENTITY;
21
22  ARCHITECTURE arch OF uart_unit IS
23
24    COMPONENT axi_uartlite_0
25      PORT (
26        s_axi_aclk : IN STD_LOGIC;
27        s_axi_aresetn : IN STD_LOGIC;
28        interrupt : OUT STD_LOGIC;
29        s_axi_awaddr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
30        s_axi_awvalid : IN STD_LOGIC; --address valid
31        s_axi_awready : OUT STD_LOGIC;
32        s_axi_wdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
33        s_axi_wstrb : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
34        s_axi_wvalid : IN STD_LOGIC;
35        s_axi_wready : OUT STD_LOGIC;
```

```
36        s_axi_bresp : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
37        s_axi_bvalid : OUT STD_LOGIC;
38        s_axi_bready : IN STD_LOGIC;
39        s_axi_araddr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
40        s_axi_arvalid : IN STD_LOGIC;
41        s_axi_arready : OUT STD_LOGIC;
42        s_axi_rdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
43        s_axi_rresp : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
44        s_axi_rvalid : OUT STD_LOGIC;
45        s_axi_rready : IN STD_LOGIC;
46        rx : IN STD_LOGIC;
47        tx : OUT STD_LOGIC
48      );
49    END COMPONENT;
50
51    SIGNAL awvalid, awready, wvalid, wready : STD_LOGIC;
52    SIGNAL wdata : STD_LOGIC_VECTOR(31 DOWNTO 0);
53    SIGNAL wstrb, awaddr : STD_LOGIC_VECTOR(3 DOWNTO 0);
54    TYPE state_t IS (IDLE, INIT, WAIT_ACK, SET_WRITE, WRITE_UART,
      ↪  CHECK_FIFO_FULL, CHECK_FIFO_FULL_2, GET_DATA, DEBUG);
55    SIGNAL state, state_nxt : state_t;
56    SIGNAL bvalid : STD_LOGIC;
57    SIGNAL count, count_nxt : NATURAL;
58    SIGNAL bresp : STD_LOGIC_VECTOR(1 DOWNTO 0);
59    SIGNAL arvalid, arready : STD_LOGIC;
60    SIGNAL araddr : STD_LOGIC_VECTOR(3 DOWNTO 0);
61    SIGNAL rdata : STD_LOGIC_VECTOR(32 - 1 DOWNTO 0);
62    SIGNAL wrdata_reg, wrdata_reg_nxt : STD_LOGIC_VECTOR(7 DOWNTO 0);
63  BEGIN
64
65    sync : PROCESS (clk, reset)
66      VARIABLE var : NATURAL := 1;
67    BEGIN
68      IF reset = '0' THEN
69        state <= IDLE;
70        count <= 0;
71        wrdata_reg <= (OTHERS => '0');
72      ELSIF rising_edge(clk) THEN
73        state <= state_nxt;
74        count <= count_nxt;
75        wrdata_reg <= wrdata_reg_nxt;
76      END IF;
77    END PROCESS;
78
79    state_p : PROCESS (ALL)
80      VARIABLE debug_count : NATURAL := 0;
81    BEGIN
82      awvalid <= '0';
83      awaddr <= "0000";
```

```
84      wstrb <= "0000";
85      state_nxt <= state;
86      wvalid <= '0';
87      wdata <= (OTHERS => '0');
88      count_nxt <= count;
89      arvalid <= '0';
90      wrdata_reg_nxt <= wrdata_reg;
91      from_uart.ready <= '0';
92      araddr <= "0000";
93      CASE (state) IS
94        WHEN IDLE =>
95          state_nxt <= INIT;
96        WHEN INIT =>
97          awaddr <= "1100";
98          awvalid <= '1';
99          wstrb <= "0010";
100         wdata(8 - 1 DOWNTO 0) <= X"13";
101         wvalid <= '1';
102         IF wready = '1' THEN
103           state_nxt <= WAIT_ACK;
104         END IF;
105
106       WHEN WAIT_ACK =>
107         awvalid <= '0';
108         wstrb <= "0010";
109         wdata(8 - 1 DOWNTO 0) <= X"13";
110         wvalid <= '0';
111         IF bvalid = '1' THEN
112           state_nxt <= GET_DATA;
113           -- state_nxt <= DEBUG;
114         END IF;
115
116         --   when DEBUG =>
117         --    from_uart.ready <= '1';
118         --    state_nxt <= DEBUG;
119         --    if debug_count = 10000 then
120         --        from_uart.ready <= '0';
121         --    else
122         --        debug_count := debug_count +1;
123         --    end if;
124
125       WHEN GET_DATA =>
126         from_uart.ready <= '1';
127         IF to_uart.valid = '1' THEN
128           state_nxt <= SET_WRITE;
129           wrdata_reg_nxt <= to_uart.data;
130         END IF;
131
132       WHEN SET_WRITE =>
```

```
133           awaddr <= "0100";
134           awvalid <= '1';
135           wdata(8 - 1 DOWNTO 0) <= wrdata_reg_nxt;
136
137           wvalid <= '1';
138           wstrb <= "0000";
139           IF wready = '1' THEN
140             count_nxt <= count + 1;
141             state_nxt <= WRITE_UART;
142           END IF;
143
144         WHEN WRITE_UART =>
145           awvalid <= '0';
146           wvalid <= '0';
147           IF bresp = "00" THEN
148             state_nxt <= CHECK_FIFO_FULL;
149           END IF;
150
151         WHEN CHECK_FIFO_FULL =>
152           araddr <= "1000";
153           arvalid <= '1';
154           IF arready = '1' THEN
155             state_nxt <= CHECK_FIFO_FULL_2;
156           END IF;
157
158         WHEN CHECK_FIFO_FULL_2 =>
159           IF rdata(3) = '0' THEN
160             state_nxt <= GET_DATA;
161           ELSE
162             state_nxt <= CHECK_FIFO_FULL;
163           END IF;
164         WHEN OTHERS =>
165       END CASE;
166     END PROCESS;
167     out_p : PROCESS (ALL)
168     BEGIN
169       from_uart.want_data_ofm <= '0';
170       from_uart.want_data_counters <= '0';
171       IF finished = '1' THEN
172         IF finished_counters = '1' THEN
173
174           from_uart.want_data_ofm <= '1';
175         ELSE
176           from_uart.want_data_counters <= '1';
177         END IF;
178       END IF;
179     END PROCESS;
180     uartlite_i : axi_uartlite_0
181     PORT MAP(
```

```
182       s_axi_aclk => clk,
183       s_axi_aresetn => reset,
184       interrupt => OPEN,
185       s_axi_awaddr => awaddr,
186       s_axi_awvalid => awvalid,
187       s_axi_awready => awready,
188       s_axi_wdata => wdata,
189       s_axi_wstrb => wstrb,
190       s_axi_wvalid => wvalid,
191       s_axi_wready => wready,
192       s_axi_bresp => bresp,
193       s_axi_bvalid => bvalid,
194       s_axi_bready => '1',
195       s_axi_araddr => araddr,
196       s_axi_arvalid => arvalid,
197       s_axi_arready => arready,
198       s_axi_rdata => rdata,
199       s_axi_rresp => OPEN,
200       s_axi_rvalid => OPEN,
201       s_axi_rready => '1',
202       rx => rx,
203       tx => tx
204     );
205 
206 END ARCHITECTURE;
```

B.2 Python

scripts/deepLabv3main.py

```
1  from tqdm import tqdm
2  import network
3  import utils
4  import os
5  import random
6  import argparse
7  import numpy as np
8 
9  import copy
10 
11 from torch.utils import data
12 from datasets import VOCSegmentation, Cityscapes
13 from utils import ext_transforms as et
14 from metrics import StreamSegMetrics
15 
16 import torch
17 import torch.nn as nn
18 from utils.visualizer import Visualizer
```

```
19
20  from PIL import Image
21  import matplotlib
22  import matplotlib.pyplot as plt
23
24  import torch.nn.utils.prune as prune
25  import torch.nn.functional as F
26
27  import torch.quantization
28
29  import warnings
30  warnings.filterwarnings(
31      action='ignore',
32      category=DeprecationWarning,
33      module=r'.*'
34  )
35  warnings.filterwarnings(
36      action='default',
37      module=r'torch.quantization'
38  )
39
40
41  def get_argparser():
42      parser = argparse.ArgumentParser()
43
44      # Datset Options
45      parser.add_argument("--data_root", type=str,
        ↪  default='./datasets/data',
46                          help="path to Dataset")
47      parser.add_argument("--dataset", type=str, default='voc',
48                          choices=['voc', 'cityscapes'], help='Name of
                            ↪  dataset')
49      parser.add_argument("--num_classes", type=int, default=None,
50                          help="num classes (default: None)")
51
52      # Deeplab Options
53      parser.add_argument("--model", type=str,
        ↪  default='deeplabv3plus_mobilenet',
54                          choices=['deeplabv3_resnet50',
                            ↪  'deeplabv3plus_resnet50',
55                                   'deeplabv3_resnet101',
                                     ↪  'deeplabv3plus_resnet101',
56                                   'deeplabv3_mobilenet',
                                     ↪  'deeplabv3plus_mobilenet'],
                                     ↪  help='model name')
57      parser.add_argument("--separable_conv", action='store_true',
        ↪  default=False,
58                          help="apply separable conv to decoder and
                            ↪  aspp")
```

```
59      parser.add_argument("--output_stride", type=int, default=16,
        ↪  choices=[8, 16])
60
61      # Train Options
62      parser.add_argument("--test_only", action='store_true',
        ↪  default=False)
63      parser.add_argument("--save_val_results", action='store_true',
        ↪  default=False,
64                          help="save segmentation results to
                            ↪  \"./results\"")
65      parser.add_argument("--total_itrs", type=int, default=30e3,
66                          help="epoch number (default: 30k)")
67      parser.add_argument("--lr", type=float, default=0.01,
68                          help="learning rate (default: 0.01)")
69      parser.add_argument("--lr_policy", type=str, default='poly',
        ↪  choices=['poly', 'step'],
70                          help="learning rate scheduler policy")
71      parser.add_argument("--step_size", type=int, default=10000)
72      parser.add_argument("--crop_val", action='store_true',
        ↪  default=False,
73                          help='crop validation (default: False)')
74      parser.add_argument("--batch_size", type=int, default=16,
75                          help='batch size (default: 16)')
76      parser.add_argument("--val_batch_size", type=int, default=4,
77                          help='batch size for validation (default:
                            ↪  4)')
78      parser.add_argument("--crop_size", type=int, default=513)
79
80      parser.add_argument("--ckpt", default=None, type=str,
81                          help="restore from checkpoint")
82      parser.add_argument("--continue_training", action='store_true',
        ↪  default=False)
83      parser.add_argument("--extract_values", action='store_true',
        ↪  default=False)
84      parser.add_argument("--pruning_rate", type=float, default=0.1)
85      parser.add_argument("--loss_type", type=str,
        ↪  default='cross_entropy',
86                          choices=['cross_entropy', 'focal_loss'],
                            ↪  help="loss type (default: False)")
87      parser.add_argument("--gpu_id", type=str, default='0',
88                          help="GPU ID")
89      parser.add_argument("--weight_decay", type=float, default=1e-4,
90                          help='weight decay (default: 1e-4)')
91      parser.add_argument("--random_seed", type=int, default=1,
92                          help="random seed (default: 1)")
93      parser.add_argument("--print_interval", type=int, default=10,
94                          help="print interval of loss (default: 10)")
95      parser.add_argument("--val_interval", type=int, default=100,
```

```
                        help="epoch interval for eval (default:
                        ↪  100)")
    parser.add_argument("--download", action='store_true',
    ↪  default=False,
                        help="download datasets")

    # PASCAL VOC Options
    parser.add_argument("--year", type=str, default='2012',
                        choices=['2012_aug', '2012', '2011', '2009',
                        ↪  '2008', '2007'], help='year of VOC')

    # Visdom options
    parser.add_argument("--enable_vis", action='store_true',
    ↪  default=False,
                        help="use visdom for visualization")
    parser.add_argument("--vis_port", type=str, default='13570',
                        help='port for visdom')
    parser.add_argument("--vis_env", type=str, default='main',
                        help='env for visdom')
    parser.add_argument("--vis_num_samples", type=int, default=8,
                        help='number of samples for visualization
                        ↪  (default: 8)')
    return parser

iter_extraction = 0

def get_dataset(opts):
    """ Dataset And Augmentation
    """
    if opts.dataset == 'voc':
        train_transform = et.ExtCompose([
            #et.ExtResize(size=opts.crop_size),
            et.ExtRandomScale((0.5, 2.0)),
            et.ExtRandomCrop(size=(opts.crop_size, opts.crop_size),
            ↪  pad_if_needed=True),
            et.ExtRandomHorizontalFlip(),
            et.ExtToTensor(),
            et.ExtNormalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225]),
        ])
        if opts.crop_val:
            val_transform = et.ExtCompose([
                et.ExtResize(opts.crop_size),
                et.ExtCenterCrop(opts.crop_size),
                et.ExtToTensor(),
                et.ExtNormalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225]),
            ])
        else:
```

```
139             val_transform = et.ExtCompose([
140                 et.ExtToTensor(),
141                 et.ExtNormalize(mean=[0.485, 0.456, 0.406],
142                                 std=[0.229, 0.224, 0.225]),
143             ])
144         train_dst = VOCSegmentation(root=opts.data_root,
            ↪  year=opts.year,
145                                     image_set='train',
                                        ↪  download=opts.download,
                                        ↪  transform=train_transform)
146         val_dst = VOCSegmentation(root=opts.data_root,
            ↪  year=opts.year,
147                                   image_set='val', download=False,
                                      ↪  transform=val_transform)
148
149     if opts.dataset == 'cityscapes':
150         train_transform = et.ExtCompose([
151             #et.ExtResize( 512 ),
152             et.ExtRandomCrop(size=(opts.crop_size, opts.crop_size)),
153             et.ExtColorJitter( brightness=0.5, contrast=0.5,
                ↪  saturation=0.5 ),
154             et.ExtRandomHorizontalFlip(),
155             et.ExtToTensor(),
156             et.ExtNormalize(mean=[0.485, 0.456, 0.406],
157                             std=[0.229, 0.224, 0.225]),
158         ])
159
160         val_transform = et.ExtCompose([
161             #et.ExtResize( 512 ),
162             et.ExtToTensor(),
163             et.ExtNormalize(mean=[0.485, 0.456, 0.406],
164                             std=[0.229, 0.224, 0.225]),
165         ])
166
167         train_dst = Cityscapes(root=opts.data_root,
168                                split='train',
                                   ↪  transform=train_transform)
169         val_dst = Cityscapes(root=opts.data_root,
170                              split='val', transform=val_transform)
171     return train_dst, val_dst
172
173
174 def validate(opts, model, loader, device, metrics,
    ↪  ret_samples_ids=None):
175     """Do validation and return specified samples"""
176     metrics.reset()
177     ret_samples = []
178     if opts.save_val_results:
179         if not os.path.exists('results'):
```

```
            os.mkdir('results')
        denorm = utils.Denormalize(mean=[0.485, 0.456, 0.406],
                                   std=[0.229, 0.224, 0.225])
        img_id = 0

    with torch.no_grad():
        for i, (images, labels) in tqdm(enumerate(loader)):

            images = images.to(device, dtype=torch.float32)
            labels = labels.to(device, dtype=torch.long)

            outputs = model(images)
            preds = outputs.detach().max(dim=1)[1].cpu().numpy()
            targets = labels.cpu().numpy()

            metrics.update(targets, preds)
            if ret_samples_ids is not None and i in ret_samples_ids:
            ↪  # get vis samples
                ret_samples.append(
                    (images[0].detach().cpu().numpy(), targets[0],
                    ↪  preds[0]))

            if opts.save_val_results:
                for i in range(len(images)):
                    image = images[i].detach().cpu().numpy()
                    target = targets[i]
                    pred = preds[i]

                    image = (denorm(image) * 255).transpose(1, 2,
                    ↪  0).astype(np.uint8)
                    target =
                    ↪  loader.dataset.decode_target(target).astype(np.uint8)
                    pred =
                    ↪  loader.dataset.decode_target(pred).astype(np.uint8)


                    ↪  Image.fromarray(image).save('results/%d_image.png'
                    ↪  % img_id)

                    ↪  Image.fromarray(target).save('results/%d_target.png'
                    ↪  % img_id)
                    Image.fromarray(pred).save('results/%d_pred.png'
                    ↪  % img_id)

                    fig = plt.figure()
                    plt.imshow(image)
                    plt.axis('off')
                    plt.imshow(pred, alpha=0.7)
                    ax = plt.gca()
```

```
                        ax.xaxis.set_major_locator(matplotlib.ticker.NullLocator())
                        ax.yaxis.set_major_locator(matplotlib.ticker.NullLocator())
                        plt.savefig('results/%d_overlay.png' % img_id, bbox_inches='tight', pad_inches=0)
                        plt.close()
                        img_id += 1

        score = metrics.get_results()
    return score, ret_samples

def quantization(model,val_loader):
        model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
        torch.backends.quantized.engine = 'fbgemm'
        #inserting  observers
        torch.quantization.prepare(model,inplace=True)
        run(model,val_loader,2)
        torch.quantization.convert(model,inplace=True)
        print("Finished Calibration")
        return model

def run(model,loader,pos):
    model.eval()
    with torch.no_grad():
        for i, (images, labels) in tqdm(enumerate(loader)):
            #print("Run Model")
            #print("I: "+str(i))
            print(images)
            outputs = model(images)
            #print("Finished Model")
            if i == pos:
                break

def main():
    def extract_values(self, input, output):
        global iter_extraction
        print('Inside ' + self.__class__.__name__ + ' forward')
        ins = input[0].int_repr().long().detach().numpy()
        print(ins.shape)
        zero_point_in = np.array(input[0].q_zero_point())
        scale_i = np.array(input[0].q_scale())
        w = self.weight().int_repr().long().detach().numpy()
        weight_spars = (len(w.flatten())-np.count_nonzero(w.flatten()))/len(w.flatten())
        print("weight spars: "+str(weight_spars))
        spars = (len(ins.flatten())-np.count_nonzero(ins.flatten()==zero_point_in))/le
```

```
262
263             print("input spars: "+ str(spars))
264
265             ins = input[0].int_repr().long().detach().numpy()
266             #print(ins)
267             with
                ↪  open('data/input_prunned{:.2f}_{}.npy'.format(spars,iter_extraction),
                ↪  'wb') as f:
268                 np.save(f,ins)
269                 np.save(f,scale_i)
270                 np.save(f,zero_point_in)
271             w = self.weight().int_repr().long().detach().numpy()
272             scales_w =
                ↪  self.weight().q_per_channel_scales().detach().numpy()
273             zero_point_w =
                ↪  self.weight().q_per_channel_zero_points().detach().numpy()
274
275             with
                ↪  open('data/weights_prunned{:.1f}.npy'.format(weight_spars),'wb')
                ↪  as f:
276                np.save(f,w)
277                np.save(f,scales_w)
278                np.save(f, zero_point_w)
279
280
281             outs = output.int_repr().detach().numpy()
282             scale_o = np.array(output.q_scale())
283             z_o = np.array(output.q_zero_point())
284
285             with
                ↪  open('data/outputs_prunned{:.2f}_{}.npy'.format(spars,iter_extraction),
                ↪  'wb') as f:
286                 np.save(f,outs)
287                 np.save(f,scale_o)
288                 np.save(f,z_o)
289
290
291             iter_extraction += 1
292
293     opts = get_argparser().parse_args()
294     if opts.dataset.lower() == 'voc':
295         opts.num_classes = 21
296     elif opts.dataset.lower() == 'cityscapes':
297         opts.num_classes = 19
298
299     # Setup visualization
300     vis = Visualizer(port=opts.vis_port,
301                      env=opts.vis_env) if opts.enable_vis else None
302     if vis is not None:  # display options
```

```
303             vis.vis_table("Options", vars(opts))
304 
305         if (opts.extract_values):
306             device = torch.device('cpu')
307         else:
308             os.environ['CUDA_VISIBLE_DEVICES'] = opts.gpu_id
309             device = torch.device('cuda' if torch.cuda.is_available()
                ↪  else 'cpu')
310         print("Device: %s" % device)
311 
312         # Setup random seed
313         torch.manual_seed(opts.random_seed)
314         np.random.seed(opts.random_seed)
315         random.seed(opts.random_seed)
316 
317         # Setup dataloader
318         if opts.dataset=='voc' and not opts.crop_val:
319             opts.val_batch_size = 1
320 
321         train_dst, val_dst = get_dataset(opts)
322         train_loader = data.DataLoader(
323             train_dst, batch_size=opts.batch_size, shuffle=True,
                ↪  num_workers=2)
324         val_loader = data.DataLoader(
325             val_dst, batch_size=opts.val_batch_size, shuffle=True,
                ↪  num_workers=2)
326         print("Dataset: %s, Train set: %d, Val set: %d" %
327               (opts.dataset, len(train_dst), len(val_dst)))
328 
329         # Set up model
330         model_map = {
331             'deeplabv3_resnet50': network.deeplabv3_resnet50,
332             'deeplabv3plus_resnet50': network.deeplabv3plus_resnet50,
333             'deeplabv3_resnet101': network.deeplabv3_resnet101,
334             'deeplabv3plus_resnet101': network.deeplabv3plus_resnet101,
335             'deeplabv3_mobilenet': network.deeplabv3_mobilenet,
336             'deeplabv3plus_mobilenet': network.deeplabv3plus_mobilenet
337         }
338 
339         model = model_map[opts.model](num_classes=opts.num_classes,
            ↪  output_stride=opts.output_stride)
340         if opts.separable_conv and 'plus' in opts.model:
341             network.convert_to_separable_conv(model.classifier)
342         utils.set_bn_momentum(model.backbone, momentum=0.01)
343         print(model)
344         # Set up metrics
345         metrics = StreamSegMetrics(opts.num_classes)
346 
347         # Set up optimizer
```

```
348     optimizer = torch.optim.SGD(params=[
349         {'params': model.backbone.parameters(), 'lr': 0.1*opts.lr},
350         {'params': model.classifier.parameters(), 'lr': opts.lr},
351     ], lr=opts.lr, momentum=0.9, weight_decay=opts.weight_decay)
352     #optimizer = torch.optim.SGD(params=model.parameters(),
        ↪  lr=opts.lr, momentum=0.9, weight_decay=opts.weight_decay)
353     #torch.optim.lr_scheduler.StepLR(optimizer,
        ↪  step_size=opts.lr_decay_step, gamma=opts.lr_decay_factor)
354     if opts.lr_policy=='poly':
355         scheduler = utils.PolyLR(optimizer, opts.total_itrs,
            ↪  power=0.9)
356     elif opts.lr_policy=='step':
357         scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
            ↪  step_size=opts.step_size, gamma=0.1)
358
359     # Set up criterion
360     #criterion = utils.get_loss(opts.loss_type)
361     if opts.loss_type == 'focal_loss':
362         criterion = utils.FocalLoss(ignore_index=255,
            ↪  size_average=True)
363     elif opts.loss_type == 'cross_entropy':
364         criterion = nn.CrossEntropyLoss(ignore_index=255,
            ↪  reduction='mean')
365
366     def save_ckpt(path):
367         """ save current model
368         """
369         torch.save({
370             "cur_itrs": cur_itrs,
371             "model_state": model.module.state_dict(),
372             "optimizer_state": optimizer.state_dict(),
373             "scheduler_state": scheduler.state_dict(),
374             "best_score": best_score,
375         }, path)
376         print("Model saved as %s" % path)
377
378     utils.mkdir('checkpoints')
379     # Restore
380     best_score = 0.0
381     cur_itrs = 0
382     cur_epochs = 0
383     if opts.ckpt is not None and os.path.isfile(opts.ckpt):
384         #
            ↪  https://github.com/VainF/DeepLabV3Plus-Pytorch/issues/8#issuecomment-6056014
            ↪  @PytaichukBohdan
385         checkpoint = torch.load(opts.ckpt,
            ↪  map_location=torch.device('cpu'))
386         model.load_state_dict(checkpoint["model_state"])
387
```

```
388             model.to(device)
389             if opts.continue_training:
390                 optimizer.load_state_dict(checkpoint["optimizer_state"])
391                 scheduler.load_state_dict(checkpoint["scheduler_state"])
392                 cur_itrs = checkpoint["cur_itrs"]
393                 best_score = checkpoint['best_score']
394                 print("Training state restored from %s" % opts.ckpt)
395             print("Model restored from %s" % opts.ckpt)
396             del checkpoint  # free memory
397         else:
398             print("[!] Retrain")
399 
400             model.to(device)
401         if (not(opts.extract_values)):
402             model = nn.DataParallel(model)
403         #==========   Train Loop   ==========#
404         vis_sample_id = np.random.randint(0, len(val_loader),
            ↪  opts.vis_num_samples,
405                                           np.int32) if opts.enable_vis
                                              ↪  else None  # sample idxs
                                              ↪  for visualization
406         denorm = utils.Denormalize(mean=[0.485, 0.456, 0.406],
            ↪  std=[0.229, 0.224, 0.225])  # denormalization for ori images
407 
408         if opts.test_only:
409             model.eval()
410             val_score, ret_samples = validate(
411                 opts=opts, model=model, loader=val_loader, device=device,
                    ↪  metrics=metrics, ret_samples_ids=vis_sample_id)
412             print(metrics.to_str(val_score))
413             return
414 
415         if opts.extract_values:
416             model.eval()
417             module_ASPP = model.classifier.classifier[0].convs[1][0]
418 
                ↪  prune.l1_unstructured(module_ASPP,name='weight',amount=opts.pruning_ra
419             prune.remove(module_ASPP, 'weight')
420             model.to(device)
421             print("Quantization")
422             model = quantization(model,val_loader)
423             #just interupt after enough
424 
425             aspp = model.classifier.classifier[0].convs[1][0]
426             #print(model)
427             aspp.register_forward_hook(extract_values)
428             print("Extraction started")
429             val_score, ret_samples = validate(
```

```
            opts=opts, model=model, loader=val_loader, device=device,
            ↪  metrics=metrics, ret_samples_ids=vis_sample_id)
        return

    pruned = 1
    while pruned > 0.3:
        pruned -= 0.02
        print(pruned)
        #prun all conv2d operations
        for name, module in model.named_modules():
            if isinstance(module, torch.nn.Conv2d):

                ↪  prune.l1_unstructured(module,name='weight',amount=0.02)
        cur_epochs = 0
        interval_loss = 0
        while cur_epochs < 5: #cur_itrs < opts.total_itrs:
            # =====  Train  =====
            model.train()
            cur_epochs += 1
            for (images, labels) in train_loader:
                cur_itrs += 1

                images = images.to(device, dtype=torch.float32)
                labels = labels.to(device, dtype=torch.long)

                optimizer.zero_grad()
                outputs = model(images)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()

                np_loss = loss.detach().cpu().numpy()
                interval_loss += np_loss
                if vis is not None:
                    vis.vis_scalar('Loss', cur_itrs, np_loss)

                if (cur_itrs) % 10 == 0:
                    interval_loss = interval_loss/10
                    print("Epoch %d, Itrs %d, Loss=%f" %
                        (cur_epochs, cur_itrs, interval_loss))
                    interval_loss = 0.0

                scheduler.step()

        save_ckpt('checkpoints_pruned/pruned_%.2f_fp32.pth' %
            (pruned))
        print("validation...")
        model.eval()
        #unquantized
```

```
477             val_score, ret_samples = validate(
478                 opts=opts, model=model, loader=val_loader, device=device,
                    ↪  metrics=metrics, ret_samples_ids=vis_sample_id)
479             print(metrics.to_str(val_score))
480 
481             f = open("results.txt", "a")
482             f.write("pruned: "+str(pruned)+"\n")
483             f.write(metrics.to_str(val_score))
484             f.write("\n")
485             f.close()
486             del ret_samples
487             del val_score
488             torch.cuda.empty_cache()
489             #quantization
490             """
491             #device = torch.device('cpu')
492             model.cpu()
493             model_quant = copy.deepcopy(model)
494 
495             model_quant.to(device)
496             model_quant.eval()
497 
498             model_quant.qconfig = torch.quantization.default_qconfig
499             torch.quantization.prepare(model_quant, inplace=True)
500             #calibrate
501 
502             val_score, ret_samples = validate(
503                 opts=opts, model=model_quant, loader=val_loader,
    ↪  device=device, metrics=metrics, ret_samples_ids=vis_sample_id)
504 
505             model_quant.cpu()
506             torch.quantization.convert(model_quant, inplace=True)
507             al_score, ret_samples = validate(
508                 opts=opts, model=model_quant, loader=val_loader,
    ↪  device=device, metrics=metrics, ret_samples_ids=vis_sample_id)
509 
510             f = open("results_quant.txt", "a")
511             f.write("pruned: "+str(pruned)+"\n")
512             f.write(metrics.to_str(val_score))
513             f.write("\n")
514             f.close()
515 
516 
517             device = torch.device('cuda' if torch.cuda.is_available()
    ↪  else 'cpu')
518             model.to(device)
519            """
520             model.train()
521 
```

```
522
523
524 if __name__ == '__main__':
525     main()
```

scripts/export_data.py

```
1  #!/usr/bin/python3
2  import numpy as np
3  import sys
4  import matplotlib.pyplot as plt
5  from bitstring import Bits
6
7  SLICE_SIZE = 64
8  ifmap_zeros = 0
9  ifmap_values = 0
10 kernel_zeros = 0
11 kernel_values = 0
12
13 ZERO_MULTS = 0
14
15 with open('input_prunned.npy', 'rb') as f:
16     ifmaps = np.load(f)[0]
17     scale_ifmap = np.load(f)
18     zero_point_ifmap = np.load(f)
19 with open('outputs_prunned.npy', 'rb') as f:
20     outs = np.load(f)
21     scale_out = np.load(f)
22     zero_points_out = np.load(f)
23
24 def calc_M(M,bits):
25     M0 = M
26     n= -1
27     M_out = 0
28     while M_out < 0.5:
29         n = n + 1
30         if M0 > 1:
31             print("This should never happen M0 > 1")
32             break
33         M_out = M0 * 2**n
34     return min(int(M_out*2**(bits-1)),(2**bits-1)-1),n #because int
       ↪  !not uint!
35
36
37 def calc_start_end(kernel,rate):
38     start_ifmap = [0,0] #y,x
39     end_ifmap = [33,33]
40     start_psum = [0,0]
41     end_psum = [33,33]
```

```
42      if kernel < 3:
43          start_psum[0] = rate
44          end_ifmap[0] = 33-rate
45      elif kernel > 5:
46          start_ifmap[0] = rate
47          end_psum[0] = 33-rate
48
49      if kernel%3 == 0:
50          end_ifmap[1] = 33-rate
51          start_psum[1] = rate
52      elif kernel%3 ==2:
53          end_psum[1] = 33-rate
54          start_ifmap[1] = rate
55      return (start_ifmap,end_ifmap),(start_psum,end_psum)
56
57  ALL = 0
58  def calc_psum_kernel(kernel,ifmap_slice,weights,rate):
59      ifmap, psum = calc_start_end(kernel,rate)
60      result = np.zeros_like(ifmap_slice[0])
61      psum_x = psum[0][0]
62      psum_y = psum[0][1]
63      global ZERO_MULTS
64      global ALL
65      for x in range(ifmap[0][0],ifmap[1][0]):
66          for y in range(ifmap[0][1],ifmap[1][1]):
67              result[psum_x,psum_y] = ifmap_slice[:,x,y]@weights
68              psum_y= psum_y+1
69
70              non_zero = ifmap_slice[:,x,y]*weights
71              ZERO_MULTS += np.count_nonzero(non_zero)
72              ALL += len(non_zero)
73          psum_x = psum_x+1
74          psum_y= psum[0][1]
75      return result
76
77
78  def
    ↪  export_weights(weights,slice_size,parallel_ofm,max_ofms,filter_depth):
79      kernel_zeros = 0
80      kernel_values = 0
81      zero_points_kernels = 0
82      print("zero point kernel: "+ str(zero_points_kernels))
83      file_des = open("../data/weights.data",'w')
84      f = open("../data/kernels_mem.data",'w')
85      for ofms in range(0,max_ofms,parallel_ofm):#for ofms in
        ↪  range(0,256,parallel_ofm):
86          for filters in range(filter_depth):
87              kernel = 4
88              for y in range(0,3):
```

```
89                  for x in range(0,3):
90                      for I in range(parallel_ofm):
91                          for J in range(slice_size):
92
                                ↪  file_des.write(str(weights[I+ofms,filters*slice_size+J].
                                ↪  " ")
93      #
        ↪  print(weights[I+ofms,filters*slice_size+J].flatten()[kernel],
        ↪  end = " ")
94                              f.write(str(Bits(int =
                                ↪  int(weights[I+ofms,filters*slice_size+J].flatten()[kerne
                                ↪  = 8).bin))
95                          if
                            ↪  ((weights[I+ofms,filters*slice_size+J].flatten()[kernel])==0
96                              kernel_zeros = kernel_zeros + 1
97       #                       print('\n')
98                          kernel_values += 1
99                          file_des.write('\n')
100                         f.write('\n')
101                 kernel = kernel + 1
102                 if kernel == 9:
103                     kernel = 0
104     #add another line for the last one process
105     for i in range(parallel_ofm):
106         for y in range(slice_size):
107            file_des.write(str(0)+ " ")
108         file_des.write('\n')
109     file_des.close()
110     f.close()
111     return kernel_zeros, kernel_values
112
113
114 def export_ifmaps_bram(slice_size,depth):
115     ifmap_zeros = 0
116     ifmap_values = 0
117     with open('input_prunned.npy', 'rb') as f:
118         ifmaps = np.load(f)[0]
119         scale_ifmap = np.load(f)
120         zero_point_ifmap = np.load(f)
121
122     print("zero point ifmap: "+ str(zero_point_ifmap))
123     f = open("../data/ifmaps_mem.data",'w')
124     for filters in range(0,depth*slice_size,slice_size):
125         for y in range(0,33):
126             for x in range(0,33):
127                 i = 0
128                 for J in range(slice_size):
129                     f.write(str(Bits(uint =
                        ↪  int(ifmaps[filters+J,y,x]),length = 8).bin))
```

```
                for padd in range(8):
                    f.write(str(Bits(int = int(0),length = 8).bin))
                if (ifmaps[filters+J,y,x] == zero_point_ifmap):
                    ifmap_zeros += 1
                ifmap_values += 1
                f.write('\n')
    f.close()
    return ifmap_zeros,ifmap_values

def export_ifmaps(slice_size,depth):
    with open('input_prunned.npy', 'rb') as f:
        ifmaps = np.load(f)[0]
        scale_ifmap = np.load(f)
        zero_point_ifmap = np.load(f)

    f = open("../data/ifmaps_input.data",'w')
    for filters in range(0,depth*slice_size,slice_size):
        for y in range(0,33):
            for x in range(0,33):
                for J in range(slice_size):
                    f.write(str(ifmaps[filters+J,y,x])+ " ")
                for padd in range(8):
                    f.write(str(0)+ " ")
                f.write('\n')

    #add another line for the last one process
    for i in range(8):
        for y in range(0,slice_size):
            f.write(str(0)+ " ")
        f.write('\n')
    f.close()




def calc_ofm(weights,slice_size,depth,ofm,rate): # is true
    result = np.zeros_like(ifmaps[0].astype(int))

    for i in range(0,depth*slice_size,slice_size):
        ifmap_slice =
        ↪  ifmaps[i:i+SLICE_SIZE].astype(int)-zero_point_ifmap
        kernel = 0
        for kernel_x in range(3):
            for kernel_y in range(3):
                weights_slice =
                ↪  weights[ofm,i:i+SLICE_SIZE,kernel_x,kernel_y].astype(int)
                result = result +
                ↪  calc_psum_kernel(kernel,ifmap_slice,weights_slice,rate)
                kernel = kernel +1
```

```
176     return result
177 
178 def export_result(result,filename,op):
179     f = open(filename,op)
180     for y in range(33):
181         for x in range(33):
182             f.write(str(result[y,x].astype(int)))
183             f.write(" ")
184         f.write('\n')
185     #f.write(" ")
186     f.write('\n')
187     f.close()
188 
189 def calc_final(result,ofm,scale_weights):
190     M0 = (scale_ifmap * scale_weights[ofm]) / scale_out
191     M,n = calc_M(M0,32) #param 2 does nothing literally 0 worth
192     print(M)
193     shift = n +31
194     final = result * M
195     final = final* 2**(-shift)
196     final = np.round(final+zero_points_out)
197     return final
198 
199 def export_results(weights,slice_size,depth,ofms,scale_weights,rate):
200     op = "w"
201     for i in range(ofms):
202         result = calc_ofm(weights,slice_size,depth,i,rate)
203         export_result(result,"../data/result/result_acc.data",op)
204         result = calc_final(result,i,scale_weights)
205         export_result(result,"../data/result/result_final.data",op)
206         op = "a"
207 
208 #ofms does nothing doesnt matter
209 def export_scales(ofms,scale_weights):
210     with open('input_prunned.npy', 'rb') as f:
211         ifmaps = np.load(f)[0]
212         scale_ifmap = np.load(f)
213         zero_point_ifmap = np.load(f)
214 
215     with open('outputs_prunned.npy', 'rb') as f:
216         outs = np.load(f)
217         scale_out = np.load(f)
218         zero_points_out = np.load(f)
219     print("zero point out: "+ str(zero_points_out))
220     M0 = (scale_ifmap * scale_weights) / scale_out
221     M = np.zeros(M0.shape)
222     shift = np.zeros(M0.shape)
223     for i in range(len(M0)):
```

```
224             M[i],n= calc_M(M0[i],32) #param 2 does nothing literally 0
                ↪  worth
225             shift[i] = n +31
226
227         f = open('../data/scales.data','w')
228         for i in range(len(shift)):
229             f.write(str(Bits(int = int(M[i]),length = 32).bin))
230             f.write('\n')
231         f.close()
232
233         f = open('../data/shift.data','w')
234         for i in range(len(shift)):
235             f.write(str(Bits(uint = int(shift[i]),length = 8).bin))
236             f.write('\n')
237         f.close()
238
239
240     def reorder_weights(SLICE_SIZE, depth, ofms, weights,scale_weights):
241         ordering = []
242         for ofm in range(ofms):
243         #    print("new ofm")
244
245             var_l = []
246             for ifmap in range(0,SLICE_SIZE*depth,SLICE_SIZE):
247                 item = 0
248
249                 for x in range(3):
250                     for y in range(3):
251                         item +=
                            ↪  np.count_nonzero(weights[ofm,ifmap:ifmap+64,x,y]==0)
252                         item /= 64
253                 var_l.append(item)
254             #print(min(var_l))
255             ordering.append(min(var_l))
256         print(sorted(ordering))
257
258         fig, ax1 = plt.subplots(1, 1)
259
260         ax1.bar([i for i in range(len(ordering))],ordering)
261
262         ax1.grid(True)
263
264         plt.show()
265
266
267
268         weights_subs = weights[0:ofms]
269         scale_weights_subs = scale_weights[0:ofms]
270         ordering = np.array(ordering)
```

```
271     arr1inds = ordering.argsort()
272     weights_reorderd = weights_subs[arr1inds]
273     scales_reorderd = scale_weights_subs[arr1inds]
274     ordering = []
275     for ofm in range(ofms):
276     #    print("new ofm")
277 
278         var_l = []
279         for ifmap in range(0,SLICE_SIZE*depth,SLICE_SIZE):
280             item = 0
281 
282             for x in range(3):
283                 for y in range(3):
284                     item +=
                        ↪  np.count_nonzero(weights_reorderd[ofm,ifmap:ifmap+64,x,y]==0)
285                     item /= 64
286             var_l.append(item)
287         #print(min(var_l))
288         ordering.append(min(var_l))
289     fig, ax1 = plt.subplots(1, 1)
290 
291     ax1.bar([i for i in range(len(ordering))],ordering)
292 
293     ax1.grid(True)
294 
295     plt.show()
296     return weights_reorderd, scales_reorderd
297 
298 def calculate_ops_ideal(rate,depth,ofmaps,valid):
299     res = 33*33*64*depth*ofmaps #middle
300     res += (33-rate)*(33-rate)*64*depth*ofmaps*4#edges
301     res += (33-rate)*33*64*depth*ofmaps*4#outer middles
302     return res * valid
303 
304 
305 def export(args):
306     depth = int(args[1])
307     parallel_ofms = int(args[2])
308     ofms = int(args[3])
309     rate = int(args[4])
310 
311     print("depth="+str(depth))
312     print("OFMS="+str(ofms))
313     print("parallel="+str(parallel_ofms))
314     print("Exporting weights")
315     with open('weights_prunned.npy', 'rb') as f:
316         weights = np.load(f)
317         scale_weights = np.load(f)
318         zero_points_kernels = np.load(f)
```

```
319
320     if args[5] == "True":
321         weights, scale_weights = reorder_weights(SLICE_SIZE, depth,
            ↪  ofms, weights,scale_weights)
322         print("Reordering weights & OFMs")
323
324     w_zeros, w_values =
        ↪  export_weights(weights,SLICE_SIZE,parallel_ofms,ofms,depth)
325     print("Exporting ifmaps")
326     export_ifmaps(SLICE_SIZE,depth)
327     print("Exporting Scales")
328     if_zeros, if_values=export_ifmaps_bram(SLICE_SIZE,depth)
329     export_scales(ofms,scale_weights)
330     print("Exporting Results")
331     export_results(weights,SLICE_SIZE,depth,ofms,scale_weights,rate)
332     print("Ifmap zeros")
333     print(if_zeros/if_values)
334     print("kernel zeros")
335     print(w_zeros/w_values)
336     print("Finished")
337     print("Zero multiplications")
338     print(ZERO_MULTS/ALL)
339     print(ALL)
340     print(calculate_ops_ideal(rate,depth,ofms,1))
341 if __name__ == '__main__':
342
343     export(sys.argv)
```

scripts/convert_to_result.py

```
1  import numpy as np
2  import sys
3  from numpy import asarray
4  from numpy import save
5
6  file_in_name = sys.argv[1]
7  NUM_OFMS = int(sys.argv[4])
8  PE_COLUMNS = int(sys.argv[5])
9  PARALLEL_OFMS = int(sys.argv[3])
10 IFMAPS_DEPTH = int(sys.argv[2])
11 reorderd = sys.argv[6]
12
13 if reorderd == "True":
14     red_str = "-reorderd"
15 else:
16     red_str = ""
17 f = open(file_in_name,'rb')
18 pos = 0
19 x_offs = 0
```

```
20
21  result = np.zeros((NUM_OFMS,33,33))
22  ofm = 0
23  y = 0
24  vals = []
25
26  utilization = np.zeros((PARALLEL_OFMS,PE_COLUMNS))
27  total_cycles = 0
28
29
30  count = 0
31  col_utl = 0
32  ofm_utl = 0
33
34  #decode preamble i.e. utilization count
35  while (True):
36      count = count +1
37
38      sliced = f.read(1)
39      print(sliced)
40      int_val = int.from_bytes(sliced, byteorder='little')
41      vals.append(int_val)
42      if ((count % 4 == 0) and (count != 0)):
43          res = 0
44          for i in range(4):
45              res = res + vals[i]*16**(6-2*i)
46          vals = []
47
48          if count == 4:
49              total_cycles = res
50          else:
51              utilization[ofm_utl,col_utl]=res
52              if col_utl == PE_COLUMNS-1:
53                  col_utl = 0
54                  ofm_utl +=1
55                  if ofm_utl == PARALLEL_OFMS:
56                      break
57              else:
58                  col_utl += 1
59
60  #sliced = f.read(1)
61  print(utilization)
62  print(total_cycles)
63  elems = utilization.shape[0]*utilization.shape[1]
64  utiliz_all = np.sum(utilization/total_cycles)/elems
65  print(utiliz_all)
66  sliced = f.read(1)
67  print(sliced)
68
```

```
69
70  for ofm_offs in range(int(NUM_OFMS/PARALLEL_OFMS)):
71      for y in range(33):
72          for x_offs in range(int(33/PE_COLUMNS)):
73              for ofm in range(PARALLEL_OFMS):
74                  for x in range(PE_COLUMNS):
75                      sliced = f.read(1)
76
                        ↪  result[ofm+ofm_offs*PARALLEL_OFMS,y,x_offs*PE_COLUMNS+x]=
                        ↪  ord(sliced)
77
78  #print(result)
79
80  fw =
    ↪  open("processed/"+str(IFMAPS_DEPTH)+"-"+str(PARALLEL_OFMS)+"-"+str(NUM_OFMS)+"
81  for ofm in range(NUM_OFMS):
82      for x in range(33):
83          for y in range(33):
84              fw.write(str(int(result[ofm,x,y]))+" ")
85          fw.write('\n')
86      fw.write('\n')
87  fw.close()
88
89
90
91  with
    ↪  open("utilization/"+str(IFMAPS_DEPTH)+"-"+str(PARALLEL_OFMS)+"-"+str(NUM_OFMS)
    ↪  as f:
92      np.save(f,utilization)
93      np.save(f,total_cycles)
94      np.save(f, utiliz_all)
```