Chair of Integrated Systems
Department of Electrical and Computer Engineering
Technical University of Munich

Accelerating Semantic Image Segmentation on FPGA

Master thesis

Author: Saptarshi Mitra

Advisor: Manoj Rohit Vemparala
Supervisor: Prof. Dr.-Ing. Walter Stechele

Submission date: July 1, 2021

Abstract

From healthcare to autonomous driving, Deep Neural Networks (DNN) dominate over the traditional Computer Vision (CV) approaches in terms of accuracy and efficiency. Exponential increase of DNN applications require tremendous computation power of the underlying hardware resources. Naturally, the superior performance of DNN models comes at a cost of huge memory footprint and complex calculations. Even if Graphics Processing Units (GPU) are the main work-horse during the training of DNNs for their massive computational capabilities, they are not suitable for mobile deployments. Computations in remote cloud is also not suitable for unreliable network latency and potential security issues. The hardware, accelerating the inference at the edge, should offer flexibility of deployment due to the rapid-changing nature of this research area. Field-programmable Gate Array (FPGA) provides the best trade-off between performance, power-consumption and design flexibility.

Convolutional Neural Networks (CNN) are the state-of-the-art for computer vision applications. Semantic image segmentation is one of the most complex tasks in computer vision, providing pixel-wise annotations for complete scene understanding. For a critical application like autonomous driving, DeepLabV3+ model provides the state-of-the-art Mean Intersection-Over-Union (mIOU) for semantic segmentation on the CityScapes dataset. In this work, a fully pipelined hardware accelerator implementing novel dilated convolution was introduced. Using this accelerator, an end-end DeepLabV3+ deployment was possible on FPGA. This architecture exploits hardware optimizations like 3-D loop unrolling, memory tiling to maximize use of computational resources and provides 2.34× latency improvement with respect to the baseline architecture.

Further, a Genetic Algorithm (GA) based automated channel pruning technique was used to jointly optimize hardware usage and model accuracy. Finally, hardware awareness was incorporated in the pruning search by hardware heuristics and an accurate model of the custom accelerator. Overall a 4× speedup was observed for a 4% degradation in mIOU.

Acknowledgements

First and foremost, I would like to to thank Professor Walter Stechele for offering this interesting and challenging opportunity and all the researchers from the Chair of Integrated Systems (LIS) for allowing me to use resources whenever needed.

This work would not have been possible without tremendous support from my supervisor Manoj-Rohit Vemparala. I can not thank him enough for continuously pushing me and sharing his ideas over the past months.

I would like to thank the entire HAPPi-Net team, particularly Nael Al-Fasfous for his support. My sincere gratitude towards Pierpaolo Mori for his previous work and sharing knowledge with me. I am really thankful to my friend Sreetama Sarkar for her time and support for the development of this project.

I am forever indebted to all the friends and seniors from my home country, who inspired me to pursue Masters. This would not be possible without inspiration from Spandan Chowdhury. I would like to thank my friends from Stiftsbogen and Heiglhof for cheering me up in these trying times of the pandemic. A big thanks to childhood buddies, Anirban, Biltu, Pranay, Sandipan, Soumyadeep and Tamal for always staying beside me.

Last but not the least, I am grateful to my family, my parents Ashis and Kabita and my Grandparents for believing in me and always being a continuous source of support.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

Computer Vision has already started to disrupt many aspects of our daily life. Classically, feature extraction was human-engineered which determined the reliability of the model. In recent times, Deep Learning especially Convolutional Neural Networks (CNN) showed amazing performance and accuracy gain by extracting features from the spatial data. Its application widely varies from solving modern problems like object detection [1], video recognition [2], speech recognition [3], Natural Language Processing [4] to safety critical applications like autonomous driving [5]. In a generic Deep Learning Network, millions of trainable parameters are trained with labeled training data for a large number of training epochs. This high compute intensive operations are not suitable for general purpose computing devices like CPU (Central Processing Unit). As an example, AlexNet [6] achieved breakthrough results with 3 Fully connected and 5 Conv layers which had around 60 million trainable parameters. In addition, the number of MAC operations reach a very high value. To cater to these computation and memory hungry networks, several general puropsoe and application specific hardware accelerators have been proposed [7, 8, 9]. GPU is most favourable during training of models [10] for having enormous amount of computation cores and SIMD execution mode, favourable for highly-parallel algorithms. But often in case of inference, a lower power hungry solution is preferable such as in battery powered devices or in edge devices, where usually FPGAs [11] and ASICs [12] are used. Among these two, FPGA has better reprogrammability and suitable for hardware prototyping, also it provides best trade-off between energy-efficiency and computation capability. For the use case of autonomous driving, pixel level image classification or semantic segmentation [13] and its mobile deployment is an active area of research. In this thesis, we focused on a state-of-the-art semantic segmentation algorithm DeepLabV3+ [14] deployment with reconfigurable hardware, in our case an Arria10 GX [15] FPGA.

Primary motivation of this work is to design an accelerator with optimized memory hierarchies and dataflow, using High-Level Synthesis process which can meet the enormous compute and memory requirements of a substantially complex model like DeepLab. On the other hand, there have been significant efforts in recent years to compress the network size and minimize memory footprints without affecting much accuracy by various techniques [16]. For reducing the precision of operands, techniques like quantization [17] and weight sharing [18] have been introduced. Some

novel techniques like knowledge distillation [19], network pruning [20] helps to cut down number of operations. Network pruning is a technique to remove low-saliency weights and fine-tune the remaining weights to get a desired accuracy and weight reduction. Particularly we used a coarse-grained structured pruning having low deployment effort (channel pruning) [21] to reduce the model size as well as runtime latency. This work particularly addresses the challenge of *Co-designing Hardware and DNN model* by deriving hardware heuristics to guide the search algorithm while compressing the network. It aims to refine the pruning search further by integrating a hardware model of the designed semantic segmentation accelerator.

1.2. Problem Statement

Though FPGAs have a lot less computational capability as well as memory bandwidth compared to ASICs, they are mostly used because they offer flexibility and suitability for hardware prototyping. In literature, several novel FPGA based hardware accelerator architectures have been proposed. Among them, few focused on inference of semantic segmentation on low powered devices [22, 23]. In this work we have tried to convey the problems of a complex semantic segmentation algorithm which has residual and several special purpose blocks for intermediate processing of activations. To accelerate the segmentation process, the principal idea is to perform parallel multiply and add (MAC) operations by increasing the on-FPGA DSP utilization. For maximizing the computational resource utilization various loop unrolling techniques were considered, also fixed-point representation of operands helped to optimize the convolution operations. As the off-chip memory access is responsible for a significant percentage of total power consumption [24], we also tried to address this issue by loop tiling and efficient dilation for the segmentation network having ResNet architecture underneath. Primary concentration was to pipeline the architecture so that individual OpenCL kernels [25] can run independently while data are being fetched from memory and we get a functionally correct prediction result out of the FPGA.

In order to reduce the runtime latency of segmentation, the problems of exploiting coarse-grained channel pruning on our custom hardware were addressed. We aim at obtaining an aggressive compression performance and in this process, GA (*Genetic Algorithm*) is used as a tool to get desired compression ratios for individual layers. This work is concerned with guiding the pruning search by custom hardware and network specific heuristic strategies and facilitate an effective hardware-software co-design strategy.

1.3. Contribution

The contribution of this Master Thesis is the design of a FPGA based hardware accelerator for a Decoder-Encoder based complex state-of-the-art segmentation network and subsequent performance improvement leveraging hardware-aware Genetic Algorithm searched channel-pruning. To increase memory bandwidth, the accelerator uses a 16-bit fixed point precision for all operations. For this latency-critical and resource-constrained application, this project investigates in dilated convolution, efficient semantic segmentation and intelligent channel pruning strategies which are tightly coupled with each other.

Summing up, the core contributions proposed in this literature are as follows:

1. **Dilated Convolution in Hardware.** This work exploited dilated convolution for semantic image segmentation applications without the need of inflated kernels full of zeros using a novel technique. We are able to implement any dilated kernels irrespective of dilation ratio and fit the weight in the on-chip buffer. As we are picking selected weights directly from the DRAM depending on the dilation ratio of the layers, we get a speedup of 2.7×, 8.8×, 18.3× for dilation factor 2, 4 and 6 respectively.

2. **Efficient Segmentation Accelerator.** End-end DeepLabV3+ [14] implementation on Arria 10 GX FPGA using High-Level-Synthesis [26] supporting all special operations in encoder-decoder type network architecture. Exploited 3 dimensional unrolling and 2-D memory tiling to maximize parallel DSP computes and minimize off-chip memory access respectively. This gave upto 90% DSP utilizaton and a 2.34× latency improvement with respect to the baseline architecture with no tilling and unrolling.

3. **Operation based Chanel Pruning.** Our framework offers a configurable network architecture maintaining restrictions imposed by shortcut and identity connections of residual blocks due to pruning. This utilizes a Reinforcement-Learning based DDPG agent searched pruned network with 50% of original operations of DeepLab present. Overall latency improvement achieved is 1.43× compared to unpruned network.

4. **Hardware Aware Chanel Pruning for Custom Accelerator.** Identified hardware heuristics hampering pruning performance and incorporating these constraints in GA based pruning search, expecting a well optimized network configuration for our custom hardware. In the next step, a hardware model of our accelerator was also introduced additionally to steer a *latency driven* GA search which performed consistently better with respect to accuracy than operation based search. This latency based GA search gave a performance improvement of 4× over baseline implementation.

1.4. Organization

This thesis is organized into 6 chapters.

Chapter 2 gives the necessary background for understanding the content of this thesis. This chapter is again organized into 4 sections. It starts with a brief theory of neural networks especially CNNs, followed by model specific optimizations like pruning and quantization. Section 2.3 deals with various aspects of hardware accelerators designed to process DNNs and ends with the semantic segmentation network under study.

Chapter 3 presents the existing works in literature, relating to this thesis. It briefly touches on FPGA based hardware accelerators as well the state-of-the-art of semantic segmentation landscape.

Our approach towards the problem is illustrated in Chapter 4 with architectural description of our accelerator. Further, a hardware aware channel pruning framework introduced to get channel pruning benefits in our custom accelerator for DeepLab.

Chapter 5 provides extensive evaluation results of the baseline architecture showing the benefits of implementing architectural optimizations. Also, this chapter includes qualitative and quantitative analysis of the speedups achieved by introducing hardware model guided genetic algorithm based pruning search.

Finally, Chapter 6 concludes the thesis and suggests suitable extensions of this work that might be taken up in the future.

2. Background

The working principle of Neural Networks imitates how human brain really works. In Neural networks *perceptrons* does all the calculations and during training, it adjust themselves to minimize the *loss function* until the the network is significantly accurate. This chapter summarizes the basic concepts of Convolutional Neural Networks based inference for easier understanding of the readers in the following chapters. In the first section 2.1, motivations and arithmetic of strided convolution are discussed. Then in section 2.2 and 2.3, different optimization techniques in network level and hardware level are explained to facilitate deployment of Neural Network inference in low-powered devices. Finally, the chapter ends with a state-of-the-art semantic segmentation model, DeepLabV3+ with some unique functionalities.

2.1. Convolutional Neural Networks

Convolutional Neural Network (CNN) is a recent paradigm for solving modern problems of Machine Learning and Artificial Intelligence. It consists of multiple layers of convolution and pooling operations in between input and output layers. The deeper layers can extract features from other lower layers and eventually can produce a more robust model. Usually the Deep CNNs are trained in supervised mode and the weights of the neurons are adjusted using backpropagation. The CNN takes an image as input, then assigns weights and biases to some features or aspects of the image which helps to differentiate that particular image from another one. CNN was inspired by the optic nerves of human body and its architecture resembles with the organisation of visual cortex. If a neuron receives inputs ($\mathbf{x}$) and weights ($\mathbf{w}$) with bias b, the final output is obtained by applying Equation 2.1, where *f(•)* is a non-linear transfer function, further described in section 2.1.2.

$$y = f\left(\sum_i w_i \cdot x_i + b\right) \tag{2.1}$$

2.1.1. Motivations for using CNNs for Segmentation

A diverse range of applications like autonomous driving, remote-sensing imaging [27] derive knowledge from *scene understanding*. Complete scene understanding is a key Computer Vision problem that is being addressed with Semantic Segmentation.

Semantic Segmentation is one step up from the traditional classification problems, where a single class assigned to a whole output. Object detection and localization is another set of problem where labels are assigned along with the spatial location of those classes. Semantic Segmentation moves from this coarse inference to finer inference and makes dense prediction by attaching labels to each pixel.

This segmentation problem is being tackled with deep network architectures especially Convolutional Neural Networks which outclasses other traditional techniques in terms of efficiency and accuracy. There are mainly two advantages of using CNNs compared to fully connected networks. One is **Parameter Sharing**. The number of parameters are significantly less in CNNs, because while convolving over the input feature map, it shares the parameters and a single filter can detect features in the different part of the spatial dimension. Another advantage is **Sparsity of Connection**, because one value in the output feature map depends only on the weight projection area of input feature map but not on the entire input activation.

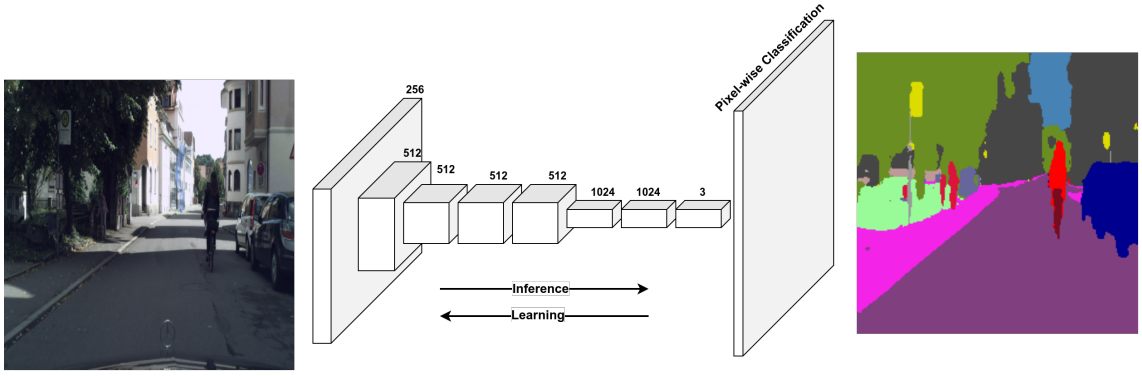

Figure 2.1.: A simple FCN network consisting of only convolutional layers, taking an image as input and gives a semantic segmented image with pixel-wise classification as output.

In the above Figure 2.1, there is an example of Fully Convolutional Network (FCN)[28], which makes pixel-wise classification of the input image of a street. On the right hand side we the expected segmented output with pixel-wise data annotation.

2.1.2. Strided Convolution Arithmetic

3-D Strided Convolution

This section lays the foundation of a simple convolution layer. In Figure 2.2, we can see a 3-D input feature map (IfMap) with dimension $N_{ix} \times N_{iy} \times N_{if}$ on the left hand over which a 4-D weight tensor with dimension $N_{kx} \times N_{ky} \times N_{if} \times N_{of}$ is strided

to get a output feature map (OfMap) of dimension $N_{ox} \times N_y \times N_{of}$. One instance of these convolutions gives a pixel in the OfMap and a number (N_{of}) of 3-D weight tensors ($N_{kx} \times N_{ky} \times N_{if}$) are strided all over the Ifmap to get N_{of} channels in the output.

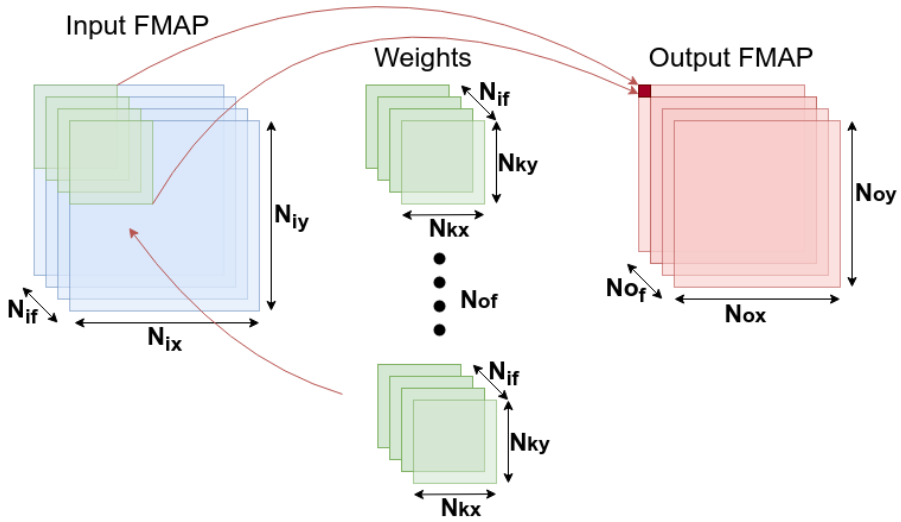

Figure 2.2.: A simple Convolutional Layer with strided Convolution. A 4-D weight tensor is required to produce 3-D ouput feature maps from a 3-D input feature map.

This strided convolution algorithm can be easily understood using the following Algorithm 1, where a 7 dimensional nested loop is responsible for a MAC operation which is the central part of a Conv layer. Here batch_size represents how many input images/activations are taken at the single time for inference.

Padding of Input Feature Maps

Depending on the dimension of the weight kernels (N_{kx} and N_{ky}) and stride value (S_x and S_y), the dimension of output feature map changes. To maintain a particular spatial dimension of OfMaps, sometimes a variable amount of zero-padding is added to the border of Ifmaps ($N_{ix} \times N_{iy}$). The output feature map dimension for non-unit stride and zero paddings (P_x and P_y) are defined by Equation 2.2:

$$N_{ox} = \left\lfloor \frac{N_{ix} - N_{kx} + 2 \cdot P_x}{S_x} \right\rfloor + 1 \; ; \; N_{oy} = \left\lfloor \frac{N_{iy} - N_{ky} + 2 \cdot P_y}{S_y} \right\rfloor + 1 \tag{2.2}$$

In most standard CNN architectures, often the size of IfMap and OfMap is kept same. A specific type of padding is applied to attain this dimension and it is called

Algorithm 1: 7D nested convolution algorithm

```
/* Loop over Batch Size                                          */
for b_i=0; b_i < batch_size; b_i++ do
    /* Loop over Output Horizontal                               */
    for out_col=0; out_col < Nox; out_col++ do
        /* Loop over Output Vertical                             */
        for out_row=0; out_row < Noy; out_row++ do
            /* Loop over Output Channel                          */
            for ch_o=0; ch_o < Nof; ch_o++ do
                /* Loop over Input Channel                       */
                for ch_i=0; ch_i < Nif; ch_i++ do
                    /* Loop over Kernel Horizontal               */
                    for i=0; i < Nkx; i++ do
                        /* Loop over Kernel Vertical             */
                        for j=0; j < Nky; j++ do
                            OfMap[batch_size][out_row][out_col][ch_o] +=
                             IfMap[batch_size][Stride*out_row+i][Stride*out_col+j][ch_i]
                             * W[i][j][ch_i][ch_o]
                        end
                    end
                end
            end
        end
    end
end
```

Same Padding or *Half Padding*. For Same Padding the padding amount should be equal to:

$$P_x = \left\lfloor \frac{N_{kx}}{2} \right\rfloor \; ; \; P_y = \left\lfloor \frac{N_{ky}}{2} \right\rfloor \tag{2.3}$$

In the below Figure 2.3, an example of same padding is demonstrated where the kernel size is 3×3 and padding is 1. So after padding the IfMaps becomes 7×7 from 5×5. But after convolution for the same zero padding, the OfMap retains the dimension 5×5.

Pooling

Pooling can be used to steadily decrease the spatial dimension of IfMap. Generally, pooling is applied in blocks and the 2-D window is slided all over input dimension just like the strided convolution. But for pooling, no weights are involved. If a

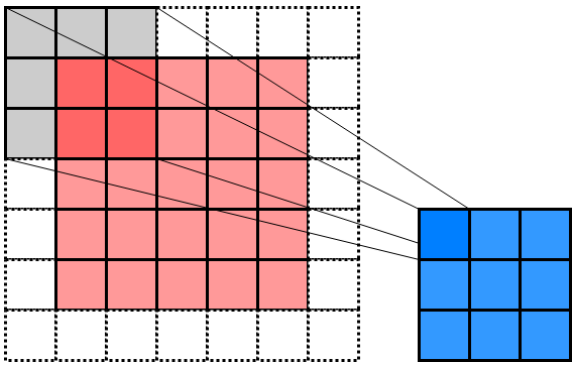

Figure 2.3.: zero padding p = 1 and non-unit stride s = 2 for a 3×3 weight convolving over a 5 × 5 padded input generating a 3 × 3 output feature map

pooling window of size $Pool_{kx} \times Pool_{ky}$ with stide S_x and S_y is applied over an IfMap of size $N_{ix} \times N_{iy}$, then the spatial dimension of OfMap is given by:

$$N_{ox} = \left\lfloor \frac{N_{ix} - Pool_{kx}}{S_x} \right\rfloor + 1 \; ; \; N_{oy} = \left\lfloor \frac{N_{iy} - Pool_{ky}}{S_y} \right\rfloor + 1 \tag{2.4}$$

While downsampling the feature maps it actually captures the features present in blocks in different areas of IfMaps. Mainly two types pooling are generally performed.

- **Max Pooling:** This is primarily used to downsample an image and get the most significant pixel values. In Fig 2.4, a 3 × 3 pooling window is applied on a IfMap of size 5 × 5. For the first stride it selects the maximum value out of the 9 values in the window, i.e. 3.

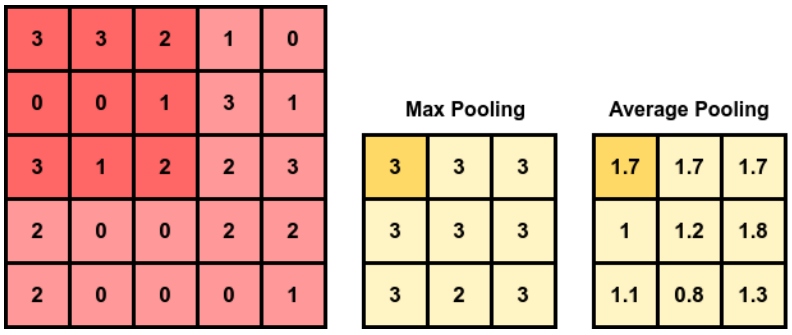

Figure 2.4.: 3 × 3 Max Pool and Average Pool is performed over a 2-dimensional input feature map having spatial dimension 5 × 5

- **Average Pooling:** Where Max Pooling extracts the most prominent features like edges, Average Pooling extracts feature more smoothly. It usually follows

a convolution layer and adds a small amount of translation invariance. In Fig 2.4, a 3×3 pooling window is applied on a IfMap of size 5×5. For the first stride it selects the average value out of the 9 values in the window, i.e. 1.7.

Activation for hidden layers

Activation function or transfer function is a elementary part of Neural Network and it defines how the weighted sum is transformed into the output. Most activation functions are non-linear and three popular choices of non-linear activation functions are plotted in Figure 2.5. The choice of activation function decides the performance of a network and controls how the network learns during the training. In our network, rectified linear activation function (ReLU) [29] is used which is the most common transfer function for hidden layers. It is very simple to implement and susceptible to *vanishing gradient* problem. ReLU simply makes the output zero if the input value is negative and in case of positive input, values are unchanged.

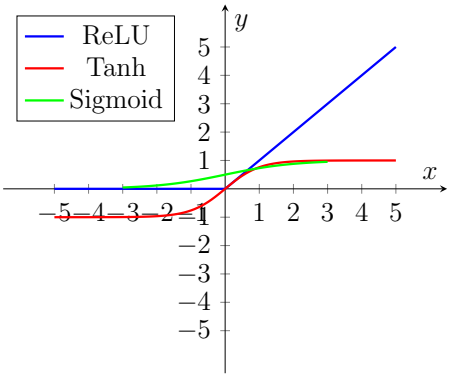

Figure 2.5.: Non-linear activation functions used in CNNs

2.1.3. Success of Deep Convolutional Networks and Residual Networks

Deep Neural Networks have gained significant popularity due to its accuracy in the most complex problems. Some very deeply stacked networks like VGG16 [30], proposed in 2014 performs really well with 92.7% top-5 test accuracy on ImageNet[31]. But training these very deep networks like Xception [32], GoogleNet [33] requires very large number of GPU hours and suffer vanishing/exploding gradient problem.

ResNet[34] was introduced in 2015, which consisted of residual blocks that helps to train the deep network. In this work, we used a state-of-the-art segmentation network (See Figure 2.9) having ResNet structure at its core. In ResNet, we have identity/skip connections which takes activation of a layer and feeds another layer deeper in the network architecture. Generally, while training a very deep network

the training error decreases at first, but after a point it increases. But the depth scaling in residual networks improve the accuracy.

2.2. Optimization of Neural Networks

The success of the Deep Neural Networks comes at a cost of high energy requirement and a huge memory footprint. The number of trainable parameters and total number of floating point operations (FLOPS) define the complexity of a deep network. Various network level techniques like **pruning** [20], **quantization** [35], **knowledge distillation** [19] have been introduced in the last decade which significantly reduces the memory requirement of these big models and makes them compatible for inference on low-power devices without loosing much accuracy.

2.2.1. Network Pruning

Not all weights of the network are equally important for the inference performance. In pruning, all of these redundancies in the neural network are carefully analyzed and discarded accordingly. After pruning, generally the remaining structure is retrained (*fine-tuning*) to recover the lost accuracy.

In pruning, only individual weights or the entire layer can be removed. On the basis of granularity of the elements removed from a network, pruning can be classified into 5 classes that are shown if Figure 2.6. Here it is interesting to notice, in case of channel pruning an entire channel from a layer is removed. This in other term means an entire weight kernel is removed from the previous layer. So channel pruning and kernel pruning term could be used interchangeably.

Also, on the basis of pruning regularities, it can be classified into 2 major classes:

- **Unstructured Pruning:** Individual weights which are redundant can be particularly removed in this pruning. There is minor or no latency degradation and easier to implement in software. But this irregular distribution of pruned weights makes it impossible for general purpose hardware to exploit the advantage and does not result performance benefit.

- **Structured Pruning:** In case of structured pruning specific kernels, filters, channels and even layers are removed from the network. From the accuracy point of view this is more challenging to retain original accuracy and leads to higher accuracy degradation but gives significant advantage while deploying in real hardware. Structured pruning, specifically channel pruning can take advantages of existing CUDA kernels [36] or in case of this accelerator, our custom OpenCL kernels to get remarkable latency benefit. It is challenging to implement channel pruning in case of residual blocks due to the presence of identity connections and the number of channels/filters should be consistent. This issue has been addressed in section 5.2.

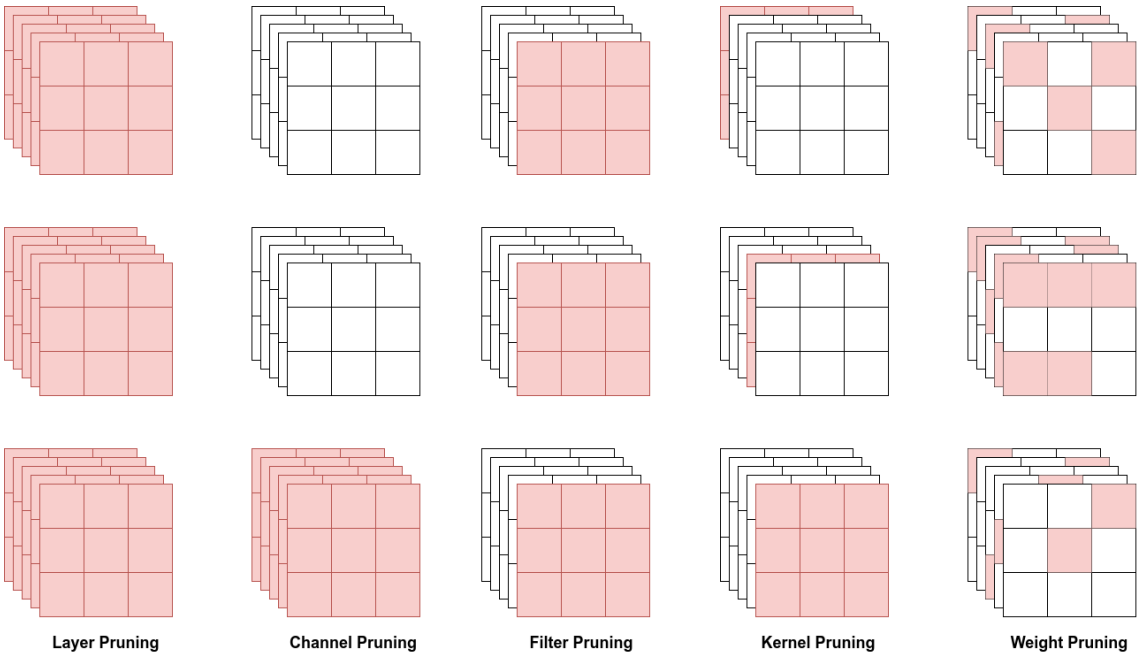

Figure 2.6.: Classification of Pruning depending on granularity of weight kernels

2.2.2. Quantization of Networks

A straightforward way to reduce model size is to use low precision operands and that is referred as *Quantization*. Primarily, there are two motivations behind quantization which yields benefit to the neural network deployment in real hardware:

1. GPU has enormous computational resources and executing all the primary operations of CNN can be performed in full precision (32/64-bit) floating point representation. This does not result any computational error in terms of loosing precision and accuracy is also not hampered. But in case of resource constrained devices at the edge, there are limited number of processing elements or DSPs available. As DSP blocks are usually data-parallel architecture, a number of MAC operations can be executed simultaneously on a single DSP if lower precision is used to represent the operands.

2. If higher number of bits used to represent an activation or weight, naturally the requirement for on-chip memory increases significantly and usually edge devices are short in memory. Also the power consumption to bring data from off-chip memory increases for full-precision networks.

Some extreme forms of quantization has been introduced in the literature [37] using 2 (Binary Nets) or 3 (Ternary Nets) unique values to represent the operands.

In case of Binary Neural Networks, the weights and activations are restricted to $\{+1, -1\}$. This representation facilitates substitution of MAC operation by a simple xnor operation followed by a counter that counts number of bits [38].

In IEEE 754 single-precision binary floating-point format there is a sign bit, a 8-bit exponent and 23-bit mantissa/fraction. The range of numbers represented by a 32-bit floating point representation is given by:

$$-2^{256-127} \leqslant x_{float} < 2^{256-127} \tag{2.5}$$

Whereas, in this thesis we used 16-bit fixed point representation for our hardware accelerator. For a N-bit fixed point representation, the number of values possible to distinctively represent is $2^N - 1$. So for a signed fixed point number we introduced a Scaling Factor (SF) that can be used to reshape the *data-range*. The range of numbers represented by a 16-bit fixed point representation is given by:

$$-2^{N-1} \cdot SF \leqslant x_{fixed} < (2^{N-1} - 1) \cdot SF \tag{2.6}$$

The reason for degradation of accuracy is mainly the resolution of the fixed-point representation. Resolution is the difference between two consecutive quantized values and it's given by $1LSB * SF$. Needless to say, the number of unique values are way lesser in fixed-point representation and conversion from floating-point number gives rise to quantization error. When rounding technique is used to truncate the precision it can result to a quantization error of $\pm\frac{1}{2}LSB \times SF$ in worst case. As the convolution operation mainly consists of multiplication and accumulation (MAC), these quantization errors are also multiplied and added up which may result a major accuracy degradation.

Another notable technique related to quantization, implemented in hardware is tackling the underflow and overflow scenario due to restricted representation range. For our accelerator we used 16-bit fixed precision numbers and for that reason the minimum and maximum numbers are capped to -32768 and 32767 for underflow (<-32768) and overflow (>32767) scenarios respectively.

2.3. Hardware Accelerator for Deep Neural Networks

In this work, the hardware acceleration of a semantic segmentation network is primarily discussed. But to understand the reason behind choosing a platform or a particular workflow, let us first see the panorama of hardware accelerators available for deployment of DNNs. The application area and primarily design constraints decides which platform is suitable for a particular problem.

2.3.1. Temporal and Spatial Architectures

The hardware processing of DNNs can be broadly classified to *temporal* and *spatial* architectures. The temporal architecture can be mostly found in CPUs and GPUs. While most CPUs can process data parallely using SIMD (Single Instruction Multiple Data), GPUs follow SIMT (Single Instruction Multiple Thread) model to maximize the huge amount of computational cores available. GPUs are the current work-horse for DNN processing especially training. As they are made for highly parallel algorithms, the parallel processing of MACs take the advantage of this. However, GPU are very power-hungry and consumes hundreds of watts. Also they are not cost effective for inference use case. So, for their fixed hardware architecture and less energy-efficiency they are not suitable for low-power mobile applications and battery-powered devices.

In case of spatial architectures, the computations take place in an array of processing elements typically connected by a NOC. Two main spatial architectures are ASICs (Application-Specific-Integrated-Circuit) and FPGAs (Field-Programmable-Gate-Arrays). ASICs are full custom chip dedicatedly developed for a particular application and they have lesser flexibility. In this rapidly moving field of research sometimes changes are needed based on the model that is being run. But at the end, these ASICs are most optimized and energy-efficient crafted for a specific application.

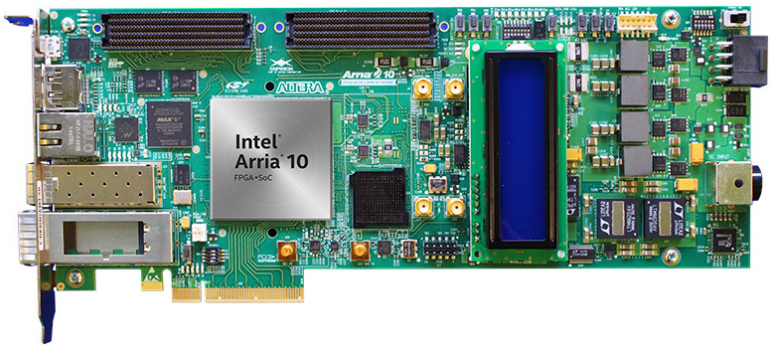

Figure 2.7.: An FPGA development kit with functional flexibility suitable for hardware prototyping

FPGAs are mainly constructed with programmable logic blocks, LUTs, DSPs, on and off-chip memories. Also there are re-configurable interconnects that wires together programmable logic blocks according to the requirement. In Figure 2.7, we see a midrange FPGA from Intel Arria device family. The FPGAs provide the best trade-off between different aspects such as energy efficiency, inference performance, programmability, and becomes a natural choice for our research. They are also really cost-effective and have a lesser complex design-flow than ASIC which helps to

get a better time-to-market. Generally FPGAs have 3-levels of memory hierarchy: DRAM, SRAM and on-chip buffers, DRAM being the most far from the processing units. More about the capacities and access latency are analyzed in chapter 4.

2.3.2. Paradigm Shift in FPGA design: High Level Synthesis

For a long time, hardware design for FPGA has been restricted to the community having sound knowledge in hardware description languages like VHDL and Verilog. In last decade, few tools have emerged that opens up the use of reconfigurable platforms to a broader skillset. Developing FPGA hardware in C++ using HLS (High-Level-Synthesis) reduces design effort significantly compared to writing RTL. Hardware blocks are generated with high level language and reiteration of design could be easily done with adding HLS constraints to get desired T (clock period). Various other vendor specific tunable settings also comes into play like use of a specific type of on-board resource or partitioning the block RAMs. Tools supplied by vendors could be used to verify area/timing. Usually the generated circuit is correct by construction, if needed co-simulation tool like ModelSim [39] can be used to verify generated RTL.

However, there are some challenges in adaptation of HLS. Usually the HLS compilers are vendor specific and the code written in high-level language must follow the coding guidelines. For example, memory access on a variable within a dynamically sized array is not possible because the moemory resource in FPGA is fixed and the HLS must know the memory requirement at the compilation time. Also, identifying parallelism can be difficult at times, depending on the algorithm complexity.

Eventhough, we decided upon OpenCL High-Level-Synthesis tool for our prototype because it is portable, open and royalty-free standard and supports virtually any devices including CPUs, GPUs and reconfigurable platforms like Arria 10 GX in our case. In Figure 2.8, we can see a OpenCL based toolflow for programming a FPGA. It provides an Application Program Interface (API) from *host* (A desktop CPU) to communicate with *device*s like FPGA over a PCIe interface and vice-versa. In the device (FPGA), some *compute units* or *kernels*, which are written using high-level C maintaining OpenCL standard, runs in parallel. These pieces of code (kernels) are pre-synthesized using a vendor-specific offline compiler and it is stored as an image file (.aocx for our case). On the other branch, the *host* application runs on a general purpose PC and manages the execution of kernels on FPGA. More about this have been explained in section 4.1.1. This host application is compiled using standard C/C++ compiler and then linked to the FPGA SDK for OpenCL runtime libraries.

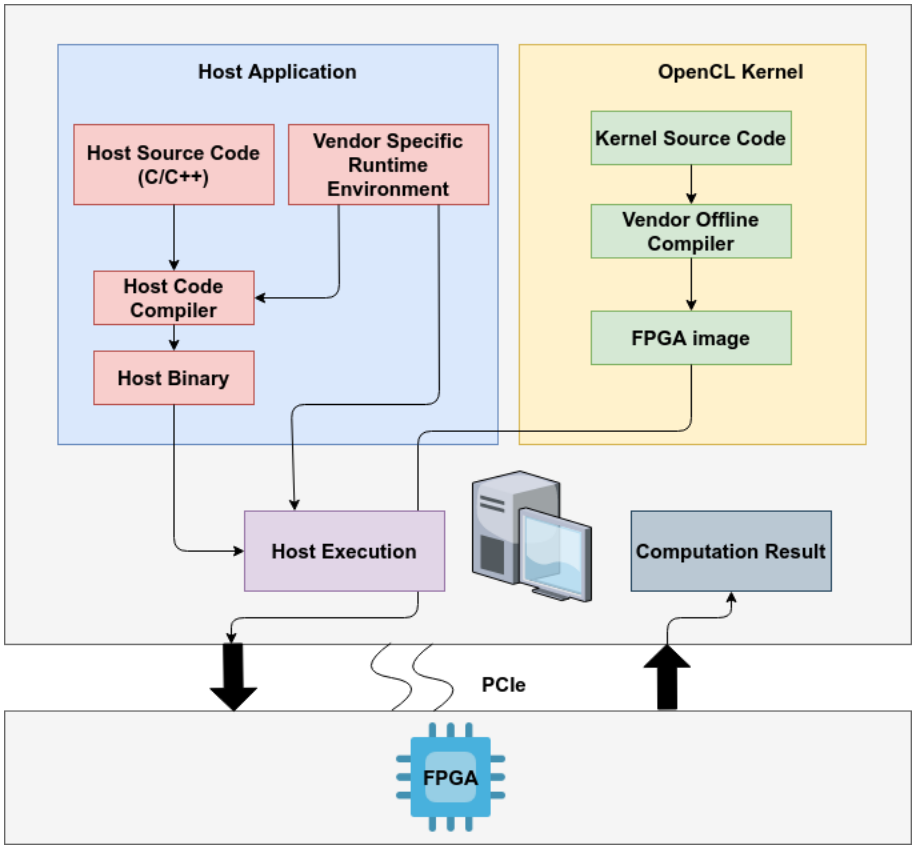

Figure 2.8.: An example toolflow of High-Level-Synthesis Framework: OpenCL

2.3.3. Optimization in Hardware Architecture

Various network and algorithmic level optimizations have been introduced in the previous section 2.2. Now some hardware architecture specific optimizations for a simple CNN layer computation, like loop unrolling and data tiling will be briefly investigated. More implementation details and experimental results for introducing these optimizations are explained in chapter 4 and 5 respectively.

Loop Unrolling

Previously for 3-dimensional strided convolution, a simple algorithm 1 has been discussed. The core of this 7-D nested loop is a MAC operation between an element of a 4-D weight tensor to an element of OfMap. But in order to exploit the parallelism in highly parallel architecture, unrolling of one or more loops can be beneficiary. The motivation behind unrolling the loops is to use as many as computational resource available to perform the MAC operations in parallel.

We use unrolling factors like P_{if}, P_{kx}, P_{ky} for unrolling along input channel, weight kernel horizontal and weight kernel vertical direction respectively. This unrolling is only possible until the PE resources are exhausted. Also logic utilization is increased to support the unrolling along multiple dimensions [40]. In the below algorithm 2, the 7-D loop has been refactored by unrolling 3 inner-loops to execute $P_{if} \times P_{kx} \times P_{ky}$ MAC operations simultaneously. Please note that the requirement of accumulator and shift-register also increases with further unrolling which has been demonstrated in Chapter 4.

Data Tiling

FPGA generally has 3 levels of memory hierarchy: DRAM, SRAM and memory buffers. These on-chip memories like SRAM are very small in size and can not accommodate the whole data volume of weight and IfMap. So many number of sparse accesses to the off-chip costs heavy in terms of power-consumption and also time consuming compared to fetching data from on-chip buffers. Numerous efforts have been found in literature, cutting down the number of external data access [41]. Data *tiling* is one of the most popular techniques, where a huge volume data is partitioned into multiple sub-volumes of pre-calculated dimension. These sub-volumes are carefully selected to be able to fit on the on-chip memory [42].

A new tiling factor (T_{ox} and T_{oy}) is introduced to the system architecture. This signifies the number of elements in the OfMap to be computed from a single tile of data. But the size of the tile buffer depends on another factor (T_{ix} and T_{iy}) which signifies the actual dimension of tile to be taken from DRAM and given by:

$$T_{ix} = (T_{ox} - 1) \cdot S_x + N_{kx} \qquad T_{iy} = (T_{oy} - 1) \cdot S_y + N_{ky} \tag{2.7}$$

If we keep on increasing the tiling factors, the size of tile buffer increases and the number of tiles needed to cover all IfMap decreases. This results a lot less number of DRAM access. But at the same time there is restriction of on-chip memory and tile buffer must be sized in a way that it can fit in the SRAM. While considering tiling the input, weight and output buffer dimension are given by the following equations:

$$IfMap_{buffer} = T_{ix} \cdot T_{iy} \cdot T_{if} \cdot sizeof(\texttt{DTYPE}) \tag{2.8}$$

$$Weight_{buffer} = N_{kx} \cdot N_{ky} \cdot T_{if} \cdot T_{of} \cdot sizeof(\texttt{DTYPE}) \tag{2.9}$$

$$OfMap_{buffer} = T_{ox} \cdot T_{oy} \cdot T_{of} \cdot sizeof(\texttt{DTYPE}) \tag{2.10}$$

If the input feature map dimension is small enough, the whole map could be covered by a single tile. The total number of tiles required to compute every pixel of OfMaps is given by the below equation 2.11. The performance improvement in

Algorithm 2: Pseudo-code of loop unrolling along P_{if}, P_{kx}, P_{ky}

```
/* Loop over Batch Size                                              */
for b_i=0; b_i < batch_size; b_i++ do
    /* Loop over Output Horizontal                                   */
    for out_col=0; out_col < Nox; out_col++ do
        /* Loop over Output Vertical                                 */
        for out_row=0; out_row < Noy; out_row++ do
            /* Loop over Output Channel                              */
            for ch_o=0; ch_o < Nof; ch_o++ do
                /* Loop over Input Channel                           */
                for ch_i=0; ch_i < Nif/Pif; ch_i++ do
                    /* Loop over Kernel Horizontal                   */
                    for i=0; i < Nkx/Pkx; i++ do
                        /* Loop over Kernel Vertical                 */
                        for j=0; j < Nky/Pky; j++ do
                            OfMap[batch_size][out_row][out_col][ch_o] +=
                              IfMap[batch_size][Stride*out_row+0][Stride*out_col+0][0]
                              * W[0][0][0][ch_o];
                            OfMap[batch_size][out_row][out_col][ch_o] +=
                              IfMap[batch_size][Stride*out_row+0][Stride*out_col+1][0]
                              * W[0][1][0][ch_o];
                            .
                            .
                            .
                            OfMap[batch_size][out_row][out_col][ch_o] +=
                              IfMap[batch_size][Stride*out_row+(i+Pkx-
                              1)][Stride*out_col+(j+Pky-1)][ch_i+Pif-1] *
                              W[i+Pkx-1][j+Pky-1][ch_i+Pif-1][ch_o]
                        end
                    end
                end
            end
        end
    end
end
```

terms of latency is available in the experiment section which indicates its success of not restricting the accelerator design memory-bandwidth bound.

$$\#TILES = \left\lceil \frac{N_{if}}{T_{if}} \right\rceil \cdot \left\lceil \frac{N_{ox}}{T_{ox}} \right\rceil \cdot \left\lceil \frac{N_{oy}}{T_{oy}} \right\rceil \cdot \left\lceil \frac{N_{of}}{T_{of}} \right\rceil \tag{2.11}$$

2.4. DeepLabV3+: A state-of-the-art Semantic Segmentation Model

With the motivation of deployment of a state-of-the-art segmentation network with our custom FPGA accelerator, we chose DeepLabV3+ [14] as our model. It is a revision of the original DeepLab architecture [43] introduced in 2016. In this architecture, authors have rethought the idea of Atrous Convolution and introduced a decoder-encoder architecture to get better segmentation result. In section 4.1.4, the hardware specific optimizations for DeepLab has been introduced. Before proceeding there, let us take a look at the elaborate architecture of the network in Figure 2.9. Here we can see a version of DeepLab network where Resnet18 acts like a backbone. The major features of this network is listed as below:

- There are 30 layers in the whole model including the shortcut connection of residual blocks.
- There is one Atrous Spatial Pyramid Pooling (ASSP) block at the end of ResNet18 which are *concatenated* channel-wise thereafter.
- There are several convolution layers in the network where dilation ratio is>1. For example, in residual block 5 there is dilation ratio 2 and in ASPP block there are several other higher dilation ratios like 6, 12 and 18 present.
- One Average pooling and two bilinear upsampling layers are present in the model which are followed by convolution layers.
- To get low-level features influencing the prediction, there is a decoder part in the model. After the upsampling ($\times 4$) of the concatenated activation of the pyramid pooling blocks they are again subjected to another channel-wise concatenation block. Output from Residual Block 2 passes through a 1×1 convolution (decoder_1 block) and that is concatenated with the activation mentioned in the last line. Please refer the Figure 2.9 for better understanding of the fairly complex network architecture.

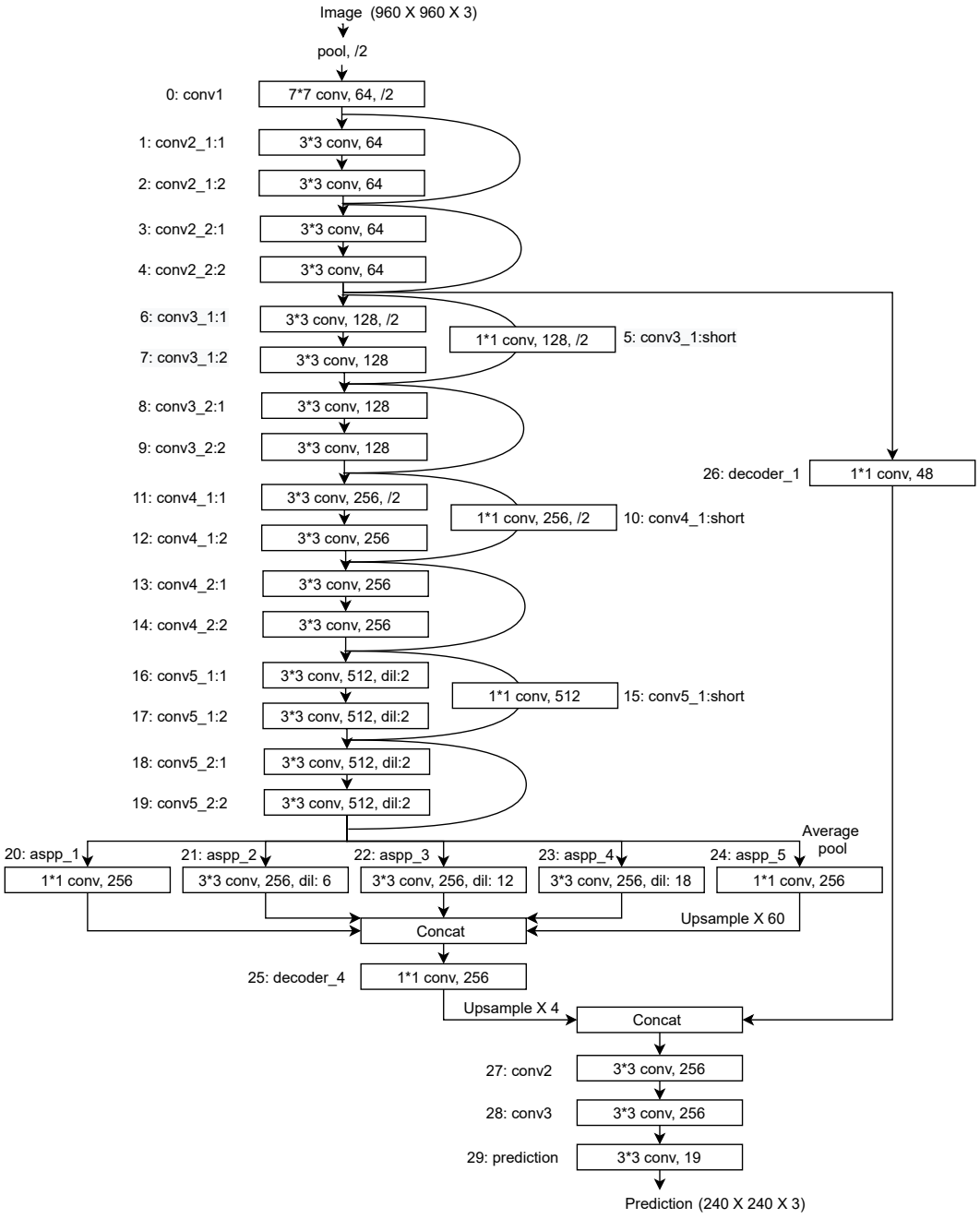

Figure 2.9.: Elaborate diagram of Deeplabv3+ Network Architecture

2.4.1. Dilated Convolution Arithmetic

Dilated convolution [44] is a specific type of strided convolution which comes from french term *atrous*, meaning porous. In this section we will explain how this special convolution works and is applied over IfMaps. The motivation behind this atrous/dilated convolution is to vary the area of projection on the IfMaps as per requirement and capture information in multiple scale.

Specifically for dilated convolution, there is an additional hyperparameter considered that controls the number of zero elements inserted in the kernel to *inflate* it. Let us refer this by D_x and D_y and setting it to 1 gives us regular convolution. When $D_x > 1$, usually $D_x - 1$ number of blank spaces or zero elements are inserted in the spatial dimension of the weight kernels to extend it. For a weight kernel with dimension N_{kx} and N_{ky} the extended dimension after dilation applied is given by:

$$\hat{N_{kx}} = N_{kx} + (N_{kx} - 1) \cdot (D_x - 1) \qquad \hat{N_{ky}} = N_{ky} + (N_{ky} - 1) \cdot (D_y - 1) \quad (2.12)$$

From Figure 2.10, we can see a simple example of Dilated Convolution with dilation rate $D_x = 2$. Naturally a 3×3 weight kernel is expanded to dimension 5×5 and it is strided over a IfMap with dimension 7×7, that results a 3×3 OfMap.

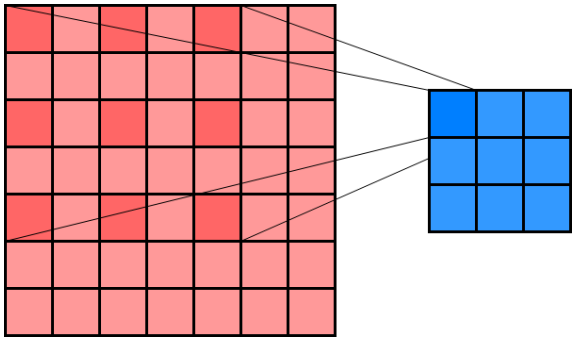

Figure 2.10.: Example of dilated convolution with dilation rate = 2 for kernel size 3×3 on input feature map dimension 7×7

The equation to calculate the spatial dimension of OfMap is modified slightly for the introduction of dilation rate hyperparameter, and is given by:

$$N_{ox} = \left\lfloor \frac{N_{ix} - N_{kx} - (N_{kx} - 1) \cdot (D_x - 1) + 2 \cdot P_x}{S_x} \right\rfloor + 1 \; ; \quad (2.13)$$

$$N_{oy} = \left\lfloor \frac{N_{iy} - N_{ky} - (N_{ky} - 1) \cdot (D_y - 1) + 2 \cdot P_y}{S_y} \right\rfloor + 1 \quad (2.14)$$

The use of dilated convolution has recently been emerged with the popularity of Networks like WaveNet [45] and DeepLab [14] where layers with varied dilation ratios are stacked up to get diverse receptive field area. However there are few hardware implementations of dilation, especially on FPGA, mainly because of high memory requirement to store extended weight kernels. This problem has been discussed in detail in section 5.1.2.

3. Related Work

This thesis presents a FPGA based hardware accelerator for semantic image segmentation. In this chapter, the research papers supporting the idea and those who inspired the work have been discussed. First in section 3.1, the foundation architecture inspiration is briefly discussed followed by some state-of-the-art architectures based on both FPGA and ASIC. The next section 3.2 gives details and motivation of semantic image segmentation along with its efficient execution from hardware community. This chapter ends with explaining some efforts to incorporate hardware specific metrics to improve channel pruning strategy.

3.1. Hardware Accelerators for CNN

A host of hardware solutions [41, 12, 46, 47] have emerged in the last decade focusing on the problem of efficient execution of DNNs especially CNNs in various use-cases. Most of them are developed as ASICs or on FPGAs with limited resources. Following are some of the most prominent works to mitigate the problems like high off-chip memory access, execution latency.

3.1.1. PipeCNN

PipeCNN [48] is one of the first works on OpenCL based hardware accelerator for CNNs that was publicly available and this was used as a baseline architecture at the beginning of our development. It consists of cascaded OpenCL kernels connected with OpenCL channel extension. This provides a deep-pipelined architecture supporting only convolution and fully connected layers. PipeCNN introduces a sliding window based data buffering scheme to cache data locally and off-chip memory bandwidth requirement are relaxed. Data vectorization and 2-D loop unrolling was implemented to increase the number of parallel MAC operations improving overall throughput. Also, the benefit of using fixed point arithmetic in place of using floating-point numbers was demonstrated reducing logical resource and memory requirement of FPGA. In our work, we extended the loop unrolling to 3-dimensions and dynamic tiling was introduced to reduce number of DRAM access. Furthermore, end-end deployment of a decoder-encoder based segmentation network was executed.

3.1.2. Intel DLA

Intel Deep Learning Accelerator (DLA) [49] published in 2017 focused maximizing the data reuse and minimizing the external memory bandwidth. Here stream buffers are used in double buffer configuration to cache the intermediate data. This methodology helps to reduce external bandwidth requirement for both conv and FC layers. They introduced a shared exponent half-precision floating point representation that permitted a lower memory footprint with almost no degradation of accuracy. A design space exploration methodology was also introduced which gave maximum possible throughput by using most computation resources possible. Furthermore, this design leverages Winograd transformation to cut down number of MAC operations. In our work, 16-bit fixed point precision was used for operands and a hardware-aware pruning technique was leveraged to optimize runtime of a segmentation network.

3.1.3. Reducing the Memory Access

Memory access being the bottleneck of CNN computation in hardware, the major focus was on investigating efficient dataflow during the development of first DNN accelerators. The mapping of operations and data movement must be well orchestrated to have significant performance and throughput. There are three kinds of data-reuse possible for a vanilla convolution layer: weight reuse (for a single channel in IfMap), input reuse (same IfMap for N_{of} times) and convolutional reuse or data tiling. The input value falling in the overlapping region of multiple strides during a convolution can be reused. In [50], an analytical framework was introduced to measure DRAM access volume determined by partitioning and scheduling configurations. For a particular network architecture, it proposes the best data-reuse scheme by providing the best tiling factor of each layer.

Based on the data reuse scheme, early introduced accelerators are classified as below:

- Weight Stationary (WS): This accelerators exploit the weight reuse and data tiling. In [51, 52] weights are stored in register files or on-chip buffers but input and partial sums are distributed.

- Output Stationary (OS): In this type of accelerators like [53][54], weights and input feature maps are spread in different ways but the partial sum after one convolution is kept fixed for a processing element (PE).

- Row Stationary (RS): This is one of the most innovative dataflows which combines the reuse of weights, inputs and partial sums. In [41], the accelerator Eyeriss showed significant performance improvement by mapping the operations of a row of a convolution to a single processing element.

In our work, we are using weight stationary dataflow alongwith convolutional reuse by exploiting tiling.

3.1.4. Exploiting Sparsity in Network

In recent years, the focus on architecture researchers are on mainly exploiting sparsity in the network and providing more flexibility to the accelerator. Usually the humongous networks are over-parameterized. With the help of advanced technique like pruning, a big part of the weights can be eliminated. Also frequent use of ReLU function makes a big part of the activation zero. If one of the weights or inputs is zero the result can be pre-computed and the entire operation could be skipped. Most accelerators use different sparsity-compression methods to make the most of this observation. The most popular schemes are : Run Length Coding (RLC), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC) and Compressed Image Size (CIS). The Cambricon-X [12] accelerator is ASIC based which appeared in 2016, exploits CIS scheme for compressing weights in asynchronous computations but does not use sparsity of the activations. ASIC based SCNN [46] uses the most advantageous CSC scheme in both weights and activations. It has a dedicated interconnection mesh to add up the scattered products. The EIE [55] accelerator also uses CSC scheme and contains extra logic to skip the zero activation values. Thus a superior energy efficiency is achieved by having less number of DRAM access. Also in FPGA, NullHop [47] is a great example of hardware accelerator using sparsity in the network. It uses the CIS scheme for weights and able to skip the null activations.

In this thesis, we used a structured pruning (channel pruning) technique to get latency benefit without dedicated HW implementation. Network specific hardware heuristics were introduced to the pruning search (using genetic algorithm) to get desirable network compression without losing much accuracy.

3.2. Accelerating Semantic Segmentation

3.2.1. Semantic Segmentation Network

For complete visual scene understanding, dividing an image to multiple meaningful sections is important and each pixel can be assigned to a specific entity. In computer vision this idea is regarded as image segmentation which actually provides pixel-wise classification for an image. As an evaluation metric for accuracy, mean Intersection over Union (mIOU) is the most widely accepted. One of the first networks providing pixel-wise dense perdition was Fully Convolutional Network (FCN) [28]. It has an entire CNN structure and can process any input size. Even though FCN demonstrated large improvement over traditional approaches it was not able to capture spatial consistency between the pixels. After the success of FCN, a host

of new encoder-decoder type of architectures became popular for semantic segmentation. As an encoder or backbone network usually successful classification models are used which produces lower-resolution images. In [56], SegNet uses max-pooling indices to upsample the feature maps in the decoder and they are further subjected to convolution to get dense predictions. With U-Net [57], in decoder part, upsampled feature maps are concatenated with cropped duplicates of OfMaps from the encoder to enhance the resolution of final output. One more trick was used to integrate context-knowledge is Conditional Random Field (CRF) [58]. In DeepLab [43], this CRF was used and it was supported by dilated convolution, expanding the receptive field to incorporate context-knowledge. Several other iterations of DeepLab have emerged since then introducing concepts like Atrous Spatial Pyramid Pooling (ASPP) along with an XCeption module [32] as decoder. This obtains the state-of-art performance for semantic segmentation on popular datasets like CityScapes [59] and Pascal-VOC 2012 [60].

3.2.2. Hardware Acceleration for Semantic Segmentation

For complex architectures and special operations there are a few efforts in accelerating the semantic segmentation networks on FPGA. [61] was one of the first works proposing efficient hardware implementation of parametric deconvolution layer. This accelerator is based on U-Net and shared memory was used between convolution and deconvolution modules. In [62], an OpenCL based flexible accelerator was introduced for decoder-encoder based architecture. The storage of the pooling indices used for unpooling operation was optimized and that saved memory consumption. It implemented SegNet-basic algorithm and achieved 48.89 GOPS throughput for CamVid dataset [63]. For remote-sensing imaging application, Liu et al. in [27] uses vector multiplication for convolution and deconvolution operations and stores intermediate feature maps in on-chip buffers. Another implementation in FPGA is found in [64], where OpenCL kernels were used for classic Encoder-Decoder semantic segmentation network SegNet. In our work we used a more complex network DeepLabV3+ [14] (with superior mean mIOU value for CityScapes or Pascal-VOC 2012), having dilation, bilinear upsampling , average pooling producing a throughput of 183.293 GOPS using almost 90% computational resource of Arria 10GX 1150 FPGA. This architecture was kept as baseline for pruning experiments to reduce latency further.

3.3. Hardware Aware Channel Pruning

Pruning is performed with an objective to reduce the hardware footprint of Machine Learning models. Automated pruning techniques tend to jointly optimize hardware metrics and model accuracy by searching for the optimum pruning configuration.

Early pruning approaches like Deep Compression by Han et al. [65] use proxy metrics like number of parameters during the search. These methods can be termed as HW agnostic since they rely on proxy metrics rather than real hardware metrics. However, it has been observed that these proxy metrics cannot accurately predict real hardware estimates. This led to hardware aware model compression, where latency or energy consumption, measured in hardware, is incorporated into the search algorithm. In AMC [66], MnasNet [67], real latency measured in smartphone devices, is considered by the RL agent while obtaining the optimum model configuration. Wang et al. in their recent paper APQ [68] construct look-up-tables, by measuring latency and energy in real hardware for different model configurations, and use it during the search. While executing the model on target hardware gives the most accurate results, this process can be time-consuming. ProxylessNAS [69] introduces hardware model for predicting model latency from layer dimensions, allowing the HW loss to be differentiable. NetAdapt [70] devises an algorithm for model compression using empirical measurements for latency so that detailed knowledge of the platform and toolchain is not required. In this thesis, we build a hardware model for our custom accelerator to get latency estimates without measuring on real hardware. While NetAdapt aims to reduce inference latency on mobile CPU or GPU, we feed the estimated latency values to our GA search algorithm with an objective to reduce inference latency on our FPGA accelerator.

4. Approach

This chapter talks about the approaches that were taken while designing a hardware accelerator for semantic image segmentation on a reconfigurable platform like FPGA. Towards the end we introduce an end-end compression pipeline using channel pruning to improve the throughput of semantic segmentation.

The first section 4.1.1 gives a generic high level architecture of the accelerator along with the functional units designed using High-Level-Synthesis tools [71]. In section 4.1.2, the generic functional units principally responsible for convolution computations are introduced followed by the essential memory buffer units to load-store intermediate weights or activations in section 4.1.3. In the next section 4.1.4, DeepLab specific functional units are presented including dilated convolution and bilinear upsampling. In the last two sections 4.2 and 4.3, a theoretical model of our accelerator is explained which helps to complete a automated channel pruning framework. This framework uses a Genetic Algorithm based search to find suitable compression ratios for a channel pruned DeepLab network.

4.1. Accelerator Design for Semantic Segmentation

4.1.1. Architectural Overview and Optimization Techniques

This accelerator consists of OpenCL kernels and the system is fully pipelined. Overall toolflow of the OpenCL framework is already explained in section 2.3.2. Network architecture and several other configurations can be configured during runtime resulting a very flexible system. The high level system diagram is shown in Figure 4.1. Overall execution of the system is managed by an application (*host* code) running independently on a general-purpose PC. Basic functionalities include:

- **Managing Data**:
  - Preprocess the image to be evaluated during inference.
  - Vectorize and devectorize data in between consecutive layer execution.
  - Allocate memory and prepare different buffers to allow device side kernels function properly.
  - Postprocess the data coming out of the FPGA at the end of the pipeline and validate against golden reference, if required.

- **Managing Kernels**:
  - In case of multiple parallel OpenCL kernels running, like ours, the host side program determines the order of kernel execution. Also it initializes and cleans up after the kernels are finished running.
  - Sets the arguments for the kernels, which they need while running.
  - Enqueue the data in the allotted buffers and transfer data to the DRAM of the FPGA. For input activation, the vector is composed of P_{if} values and for weights, it ships $P_{if} \cdot P_{kx} \cdot P_{of}$ elements to the off-chip memory. Please refer to section 2.3.3 of Background for details on unrolling factors.
- **Managing Networks**:
  - Configures the network structure that runs on the FPGA. It determines the sequence of data flowing through the kernels. For example, Max Pooling can be enabled or disabled after convolution and ReLU operation can be configured to be completed after BatchNorm or even after activations are added in case of fused layers.

While the host software routine runs, some functions are generally computationally expensive and can benefit from highly parallel processing elements. These piece of codes that are being executed independently on the FPGA are referred as *computational units*. The host program uses standard OpenCL APIs to transfer data and invoke the computational units. The OpenCL channels extension allows the computational units to talk directly with the help of FIFO buffers. It decouples data movement between simultaneously executing computational units from the host CPU. The computational units responsible for core computations are discussed in section 4.1.2 and details of DeepLabV3+ specific computational units are in section 4.1.4.

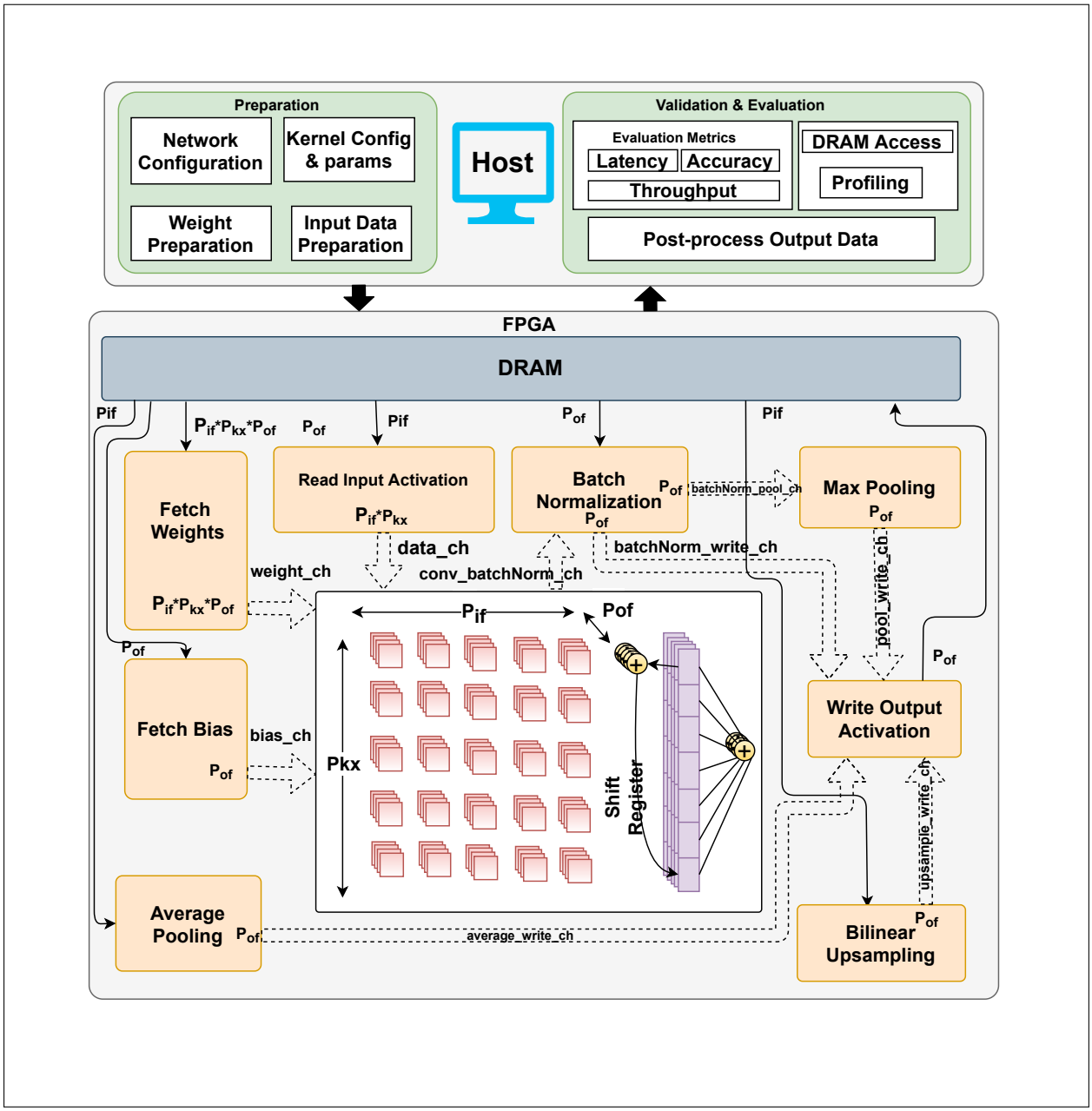

Figure 4.1.: System level architecture of the Accelerator. The green boxes are part of the host side which are running on Intel based Workstation. All the orange boxes in the lower part are standalone kernels running parallelly on the FPGA and the dotted arrows are connecting them via OpenCL channels.

4.1.2. Computational Units

In this section, the generalized computational units which are responsible for convolution, batch normalization and other relevant operations have been discussed following the optimizations mentioned in section 2.3.3. These computational units get the data from other computational units or from the memory buffer units which effectively hides the DRAM access latency.

Core Convolution in PE Array

In Figure 4.1, the central part of picture is represented by Processing Elements (PE Array) which are suitable for massive execution on parallel convolutions. In this unit, all optimizations like loop unrolling (section 2.3.3) and data tiling (section 2.3.3) have been implemented to get benefit of more parallelism and less DRAM access. According to the Arria 10 GX datasheet [15], it contains variable precision DSP blocks, which supports both fixed and floating point operations. As we used 16-bit fixed point precision for all the operations, we can execute two multiplications/DSP block because each DSP block can support two 18 x 19 multipliers or one 27 x 27 multiplier [72]. So, the number of parallel MAC operations needed are multiple of all the unrolling factors and half of that number is the DSP requirement, given by Equation 4.1.

$$\#DSP = \frac{1}{2} \cdot P_{kx} \cdot P_{if} \cdot P_{of} \tag{4.1}$$

Two functional units are responsible to bring weights and input activations from DRAM. From them, $P_{kx} \times P_{if} \times P_{of}$ and $P_{kx} \times P_{if}$ data is supplied respectively with 16-bit precision and after convolution partial sums are locally stored in a shift register with configurable depth, consecutively they are added by a adder tree. So the number of clock cycles required for one vanilla convolution layer can be represented by Equation 4.2.

$$\#CLKCycles = \left\lceil \frac{N_{if}}{P_{if}} \right\rceil \cdot \left\lceil \frac{N_{kx}}{P_{kx}} \right\rceil \cdot N_{ky} \cdot \left\lceil \frac{N_{of}}{P_{of}} \right\rceil \cdot N_{ox} \cdot N_{oy} \tag{4.2}$$

Algorithm 3 gives the *strided convolution* algorithm adapted in our custom accelerator.

Batch Normalization

Strided convolution is always followed by a Batch Normalization unit. To get reduced logic utilization, we enabled BN unit by default to apply the Scaling Factors

Algorithm 3: Convolution execution in OpenCL kernel in the accelerator

```
for out_ch_o=0; out_ch_o < ⌈N_of / P_of⌉; out_ch_o++ do
    Load Bias in the On-chip buffer
    for conv_xy=0; conv_xy < N_ox · N_oy; conv_xy++ do
        for j=0; j < ⌈N_if / P_if⌉ · ⌈N_kx / P_kx⌉ · N_ky; j++ do
            Load Weights in the On-chip buffer
            Read IFmap values
            for ll=0; ll < P_of; ll++ do
                for i=0; i < P_kx · P_if; i++ do
                    Psums[ll]= data*weight[j][ll];
                end
                Store and accumulate P_of number of Psums in shift register
            end
        end
        Produce P_of convolution output values
        Add Bias to each of P_of channels
    end
end
```

(SF) prepared for our 16-bit activations explained in section 2.2.2. This avoids repetition of scaling factor multiplication logic to the activations in the convolution unit. For each chunk of N_{of} channels coming out the computational unit for convolution, the BN is needed to be performed only once. First it is converted to floating point values to operate with other BN parameters which are directly derived from DRAM in floating-point format. From Equation 4.3, we can see how it is normalized using the parameters (mean μ, variance σ, gamma γand beta β) of the specific layer. SF_i, SF_w and SF_o are the scaling factors for input activation, weight and output activation respectively. For dummy BN layers, to keep the conv results unchanged, BN params are substituted with zeros and ones accordingly. Algorithm of the BN unit is briefly explained in 4.

$$BN_o = \texttt{int16}\left[\left(\frac{(\frac{\texttt{float32}(x)}{SF_i \cdot SF_w}) - \mu}{\sigma} \cdot \gamma - \beta\right) \cdot SF_o\right] \tag{4.3}$$

Max Pooling

Data from computational units responsible for *BatchNorm* or *CoreConv*, are received in the *MaxPool* computational unit if pooling is enabled in the network. The number of iterations largely depends on the dimension of vectorized data coming from other computational units and also on Pooling parameters like stride. Usually this unit

Algorithm 4: BatchNorm OpenCL kernel in the accelerator

for *k=0; k* $< \left\lceil \frac{N_{of}}{P_{of}} \right\rceil \cdot N_{ox} \cdot N_{oy}$ *; k++* **do**
  Read P_{of} values of convolution output from conv_batchNorm_ch
  Load P_{of} number of μ, σ, β, γfrom DRAM
  for *ll=0; ll* $< P_{of}$*; ll++* **do**
    Perform the operation of Equation 4.3
    Handles overflow and underflow cases and convert to short(16-bit)
    Performs ReLU operation if enabled in Network by the host
  end
  Write P_{of} output values to channels leading to MemWrite or MaxPool kernel
end

consists of a few compare functions which returns the maximum value of all the elements present in a selected data window. A pool buffer is filled with initial results after pooling and waits until a whole input window is received. In the current implementation the pooling unit operates row-wise and 1-D tiling along N_{ox} is supported. This is why in the first layer of DeepLab (where max pool is only enabled), 1-D tiling is applied instead of 2-D tiling.

ReLU Operation

The Rectified Linear Activation function or ReLU is a piecewise linear makes the output zero if the input is negative or it keeps the same value in output if the input is positive. It is a trivial part of neural networks as they achieve good performance and easier to train too.

ReLU operation is very computationally inexpensive. In our accelerator negative value is detected by the Most Significant Bit (MSB) in signed-number representation. Depending upon this value the ReLU operation is performed on the input activation. This activation is configurable by network, depending on where it is needed. As an example, it can performed after Conv, BatchNorm or even for fused layers after the activation from the previous layer is added.

4.1.3. Memory Buffer Units

For DNN processing on HW, the MAC operations are not the only bottleneck. DRAM access sometimes require several order of higher energy and time than the MAC operations. For each MAC operation there are 3 memory read operation and one memory write operation. If all of these accesses are through off-chip memory it can badly affect the power consumption and throughput. Because of this, it is critical to restrict the number of DRAM accesses and data movement in the memory

hierarchy as minimal as possible. Several measures have already been taken to parallelize the processing that demands a higher memory bandwidth. In this subsection, the memory buffers considered for reading or storing activation/weights/outputs in our accelerator and how they are sized have been discussed.

Weight Buffer Unit

For convolution operation activation and weights are required as operands. Clearly the weight buffer unit (*mac_weight*) storing weights is one of the most crucial buffers considered, consuming on-chip memory. For DeepLab there are some restrictions of sizing this memory, and this has been addressed in the Experiment section 5.1.2. The on-chip SRAM/M20K blocks are nearer to the computation units (hides latency), and gives a scope to reuse data over the course of convolution. From the *fetch_weights* unit weight of size $P_{kx} \times P_{if} \times P_{of}$ is fetched from DRAM and sent to Conv unit. The weight buffer is partially filled in each of $\left\lceil \frac{N_{if}}{P_{if}} \right\rceil \cdot \left\lceil \frac{N_{kx}}{P_{kx}} \right\rceil \cdot N_{ky}$ iterations and eventually it is filled with vectors suitable to get output for P_{of} channels. The total size of 4D subtensor stored at a particular time in this buffer is $N_{kx} \cdot N_{ky} \cdot N_{if} \cdot P_{of} \cdot sizeof(DTYPE)$. In our case we are using 16-bit fixed point notation for calculation and DTYPE is considered as short datatype.

Input Buffer Unit

Reading the input data from DRAM before sending to the Conv unit is done by *MemRead* unit, which has a very complex architecture. To reduce the DRAM access, it resorts to loop tiling technique elaborately described in the Background section 2.3.3.

In order to pipeline the data fetching from DRAM and sending data to Convolution unit, it uses a ***double buffering*** technique. These coupled buffers are alternatively used to fetch a whole tile from DRAM (P_{if} values at a time) and on the other hand striding over a tile window to send respective activation ($P_{kx} \times P_{if}$ values at a time) for Convolution. If tiling factors are defined as T_{ox} and T_{oy}, the input tiling factors are given by equation 4.4.

$$T_{ix} = (T_{ox} - 1) \cdot S_x + N_{kx} \qquad T_{iy} = (T_{oy} - 1) \cdot S_y + N_{ky} \tag{4.4}$$

Accordingly the on-chip buffer size is given by equation 4.5.

$$Tile\_Buffer\_Size = 2 \cdot P_{kx} \cdot \left\lceil \frac{T_{ix}}{P_{kx}} \right\rceil \cdot T_{iy} \cdot N_{if} \cdot sizeof(DataType) \tag{4.5}$$

The number of off-chip memory access for the layer, depends on the number of tiles needed to load the whole input activation and that is defined by equation 4.6.

$$\#Tiles = \left\lceil \frac{N_{ox}}{T_{ox}} \right\rceil \cdot \left\lceil \frac{N_{oy}}{T_{oy}} \right\rceil \cdot \left\lceil \frac{N_{of}}{P_{of}} \right\rceil \tag{4.6}$$

Output Buffer Unit

This buffer stores the data arriving at the end of pipeline. After that, either it is fetched again by the next layer or it is loaded by the host in case of last layer of the network. Every cycle it receives P_{of} number of results from other units (Conv/ BatchNorm/ Maxpool/ Upsampling) and it stores P_{of} number of 16-bit values to the DRAM each time.

The *MemWrite* unit has mainly two tasks:

- Storing results of the layers to the DRAM as explained above.
- In case of residual layers, the activations from a previous shortcut connection are added with the current activation. Also the fixed-point representations are converted to floating point representation in order to multiply scaling factor (SF) of the fused layer.

4.1.4. DeepLab Specific layers for Segmentation

Before going into more details of DeepLabV3+ specific units in the custom accelerator, please refer to network architecture in section 2.4. We use a version of DeepLab network where Resnet18 acts like a backbone.

Atrous Convolution [44] or Dilated Convolution is the key contribution of DeepLab based models because they are able to encode multi-scale contextual information by applying atrous convolution at multiple scales. The weight kernels are inflated with dummy zeros to get a larger area of projection on the input feature map according to the dilation ratio. The principles and arithmetic of Atrous Convolution is already explained in Background section 2.4.1 and here we will focus on the FPGA implementation. We used a novel technique to not extend the dimensions of the weight kernel but select specific activation values from the last feature map based on dilation ratio and other standard convolution parameters. Below we can see a simplfied Algorithm 5 of *MemRead* kernel, where particular features are directly loaded from the DRAM.

Concatenation and Network Restructure

ASPP was introduced in DeepLabV3 [43] after spatial pyramid pooling showed promising results in resampling features at different scales. This ASPP block consists of one 1×1 Convolution, three 3×3 Convolution with different dilation ratios and

Algorithm 5: Dilation Logic while reading data from DRAM

```
for T_i=0; k < num_tiles ; T_i++ do
    Iterate over all data vectors in the group
    for Win_i=0; Win_i < (Loop bound for DRAM < Loop bound for
     SRAM?Loop bound for SRAM:Loop bound for DRAM ); Win_i++ do
        Step 1: Load one data vector from global DRAM
        Along X direction select the features according to dilation rate, stride
         and P_kx
        Along Y direction select the features according to dilation rate and
         stride
        Step 2: Load one data vector from SRAM
    end
    replicate it P_of times and write it into data channel vector
end
```

one image level feature though averagepool, 1×1 convolution and upsampling of the last feature of ResNet18. At the end, all of these branches are concatenated and passed through a 1×1 convolution called decoder_4 as described in the Figure 2.9. In DeepLabV3+, when decoder-encoder type architecture was introduced to refine the segmentation results, another concatenation takes place joining the upsampled output of decoder_4 and another branch carrying low-level features. This channel-wise concatenation function can be executed by a standalone OpenCL kernel, but there are several implementation challenges for designing a dedicated kernel:

- **High DRAM access:** Usually the intermediate activations in between the layers are stored in on-chip memory and reused in the next layer as input. But for ASPP block, after the 5 branches are finished processing they write the output activation to the DRAM. There would have been a lot of extra off-chip memory access if they are read from the DRAM for concatenation.
- **Extra Logic Utilization:** For dedicated computational unit implementation it costs extra logic during the high-level-synthesis process. In current scenario, for the baseline accelerator we nearly used most of on-board logical resources (See Chapter 5). So we could not afford extra logic required.

Instead a novel technique was adapted to couple the ASPP concatenation and decoder_4 operation. This technique is explained for a simple layer in Figure 4.2. Here two input feature maps (IfMap1 and ifMap2) are first concatenated and then convolution is applied with weight dimension $1 \times 1 \times 2 \times 1$. In the alternative approach the weight is split in the 3rd dimension and then multiplied individually to the activations with no concatenation, now if they are added element-wise, it results the same value. Following the same approach, we structurally modified

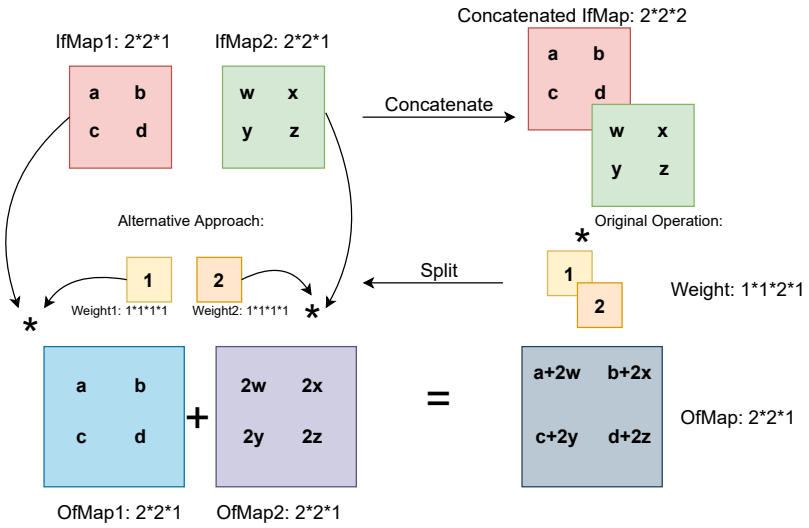

Figure 4.2.: Avoiding a dedicate kernel for concatenation followed by convolution

the DeepLabV3+ architecture to get efficient implementation in the FPGA, but functionality of the model remains unchanged. We split the decoder_4 weights into 5 parts in dimension 3 (with 256 channels in each segment) and then each of this weight segments were appended as independent 1×1 convolution layers after each of the ASPP branches. We reused the fuse layer functionality to add up the output activations and at the end of all additions we got the activation that we would have gotten after decoder_4 block. Some extra effort of pre-processing is required to prepare the weights and dummy convolution layers are appended in the network. The scaling factors (SF) are duplicated for each of the new segments.

There are two concatenations followed by convolution in the entire network, and these were replaced by multiple simple convolution layers, summed up by residual layer fusion logic. In Figure 4.3, we see a restructured DeepLabV3+ architecture, the new dummy layers are highlighted with blue and total number of layers increases from 30 to 38 (including average pooling and upsampling).

Average Pooling in ASPP Block

Global average pooling is an integral part of the ASPP block. For adopting image-level features this average pooling is applied on the last feature-map of ResNet, i.e. Res5_2:2. Independent of the spatial dimension of input feature map, this function reduces it down to 1×1 output activation with same number of output channels present. There is no weight involved in this operation. So scaling up the unit in P_{kx} direction is not relevant. The dedicated computational unit responsible for

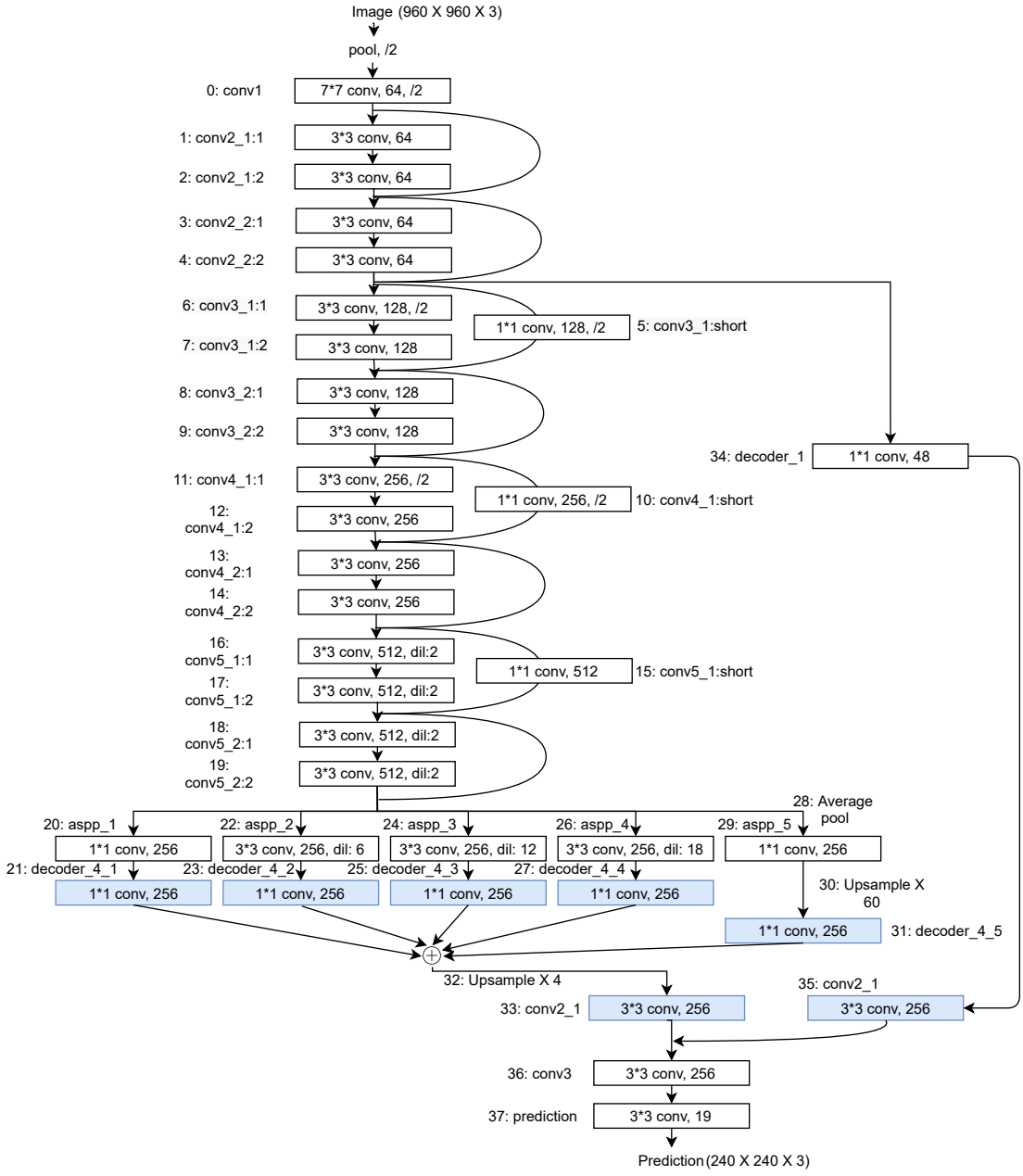

Figure 4.3.: Restructured network architecture for Deeplabv3+ to eleminate the need of dedicated concatenation kernel

Algorithm 6: Pseudo-code for Average Pooling OpenCL Kernel

```
for out_z=0; out_z < ⌈N_if / P_of⌉; out_z++ do
    for lane_div_vec=0; lane_div_vec < P_of / P_if; lane_div_vec++ do
        for i=0; i < input_xy; i++ do
            Read a vector Input vector from DRAM
            for ll=0; ll < P_if; ll++ do
                Sum up values in each channel
            end
        end
    end
end
for out_z=0; out_z < ⌈N_if / P_of⌉; out_z++ do
    for ll=0; ll < P_if; ll++ do
        Divide the sum by input_xy
        Converted to float-point values and scaling factors multiplied
        Convert back to fixed point notation
    end
    Write into the channel to MemWrite
end
```

performing average pooling directly fetch the data from DRAM, takes the average across each channel and writes the data back to DRAM via *MemWrite* unit. A dedicated input buffer is required to store the partial sums and the size of this buffer is defined by Equation 4.7. A pseudo-code of the kernel implementation can be found in Algorithm 6.

$$P\_Sum\_Size = Max(N_{of}) \cdot sizeof(DataType) \tag{4.7}$$

Bilinear Upsampling

In image processing and computer vision applications, resizing an image or activation to a specific spatial dimension is very common. In DeepLabV3+ there are 2 instances of bilinear upsampling. From Figure 2.9, we can see one upsampling is in the ASPP5 block ($1 \times 1 \longrightarrow 60 \times 60$ in spatial dimension) and the other is after decoder_4 block ($60 \times 60 \longrightarrow 240 \times 240$ in spatial dimension). Though the first one has repeated values in all of the 60×60 pixels, for the second case it is computationally expensive and the value of each of the 240×240 pixels are computed with *bilinear interpolation*. In our accelerator we designed a generic implementation of bilinear upsampling unit to address this problem which uses bilinear interpolation to calculate each of the output pixels.

Bilinear interpolation is a generalized algorithm of linear interpolation, which only works for 1D array. Let us take a look at, how interpolation works. If there are two points (a and b) having values A and B on a straight line, then the value of a third point on the line with co-ordinate x ($a \leqslant x \leqslant b$) is given by Equation 4.8:

$$X = A \cdot (1 - w) + B \cdot w \tag{4.8}$$

where

$$w = \frac{(x - a)}{(b - a)} \tag{4.9}$$

It is actually the weighted average value of the two end points. We can extend this idea to 2-D arrays and it is then called bilinear interpolation. This can be separated into two linear resizing operation in x and y direction. From Figure 4.4, we can see a simple setup where four points with values A, B, C and D are taken with coordinates (x_1, y_1), (x_2, y_1), (x_1, y_2),(x_2, y_2) respectively.

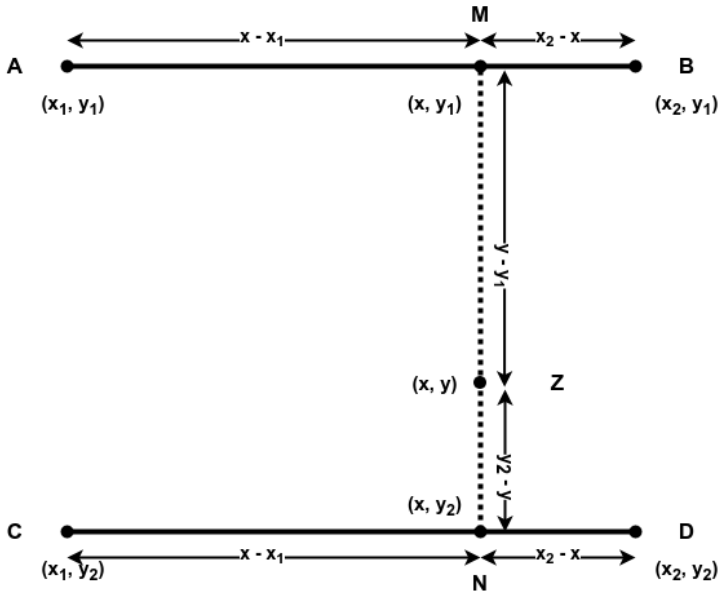

Figure 4.4.: Bilinear interpolation between 4 points in spatial dimension

First we calculate the interpolated values in the horizontal direction. The points with coordinates (x, y_1) and (x, y_2) have values M and N respectively which are defined by Equation 4.10:

$$M = A \cdot (1 - w_x) + B \cdot w_x \qquad N = C \cdot (1 - w_x) + D \cdot w_x \tag{4.10}$$

Now, we need to perform linear interpolation (in vertical direction) on the already interpolated points M and N. We get the value of Z located on the line connecting M, N with coordinate (x, y) defined by Equation 4.11:

$$Z = A \cdot (1 - w_x) \cdot (1 - w_y) + B \cdot w_x \cdot (1 - w_y) + C \cdot (1 - w_x) \cdot w_y + D \cdot w_x \cdot w_y \quad (4.11)$$

where:

$$w_x = \frac{(x - x_1)}{(x_2 - x_1)} \qquad w_y = \frac{(y - y_1)}{(y_2 - y_1)} \quad (4.12)$$

Now in our case, the idea is to find interpolated value for each output pixel. The first step is to get upsample or compression ratio between the spatial dimensions of input and output image. Then with the help of this ratio 4 pixels are selected for each output pixel to be interpolated and the weighted average is calculated. At any time during the computation of upsampling, a dedicated memory is required to store the spatial activations of P_{of} channels. The size of this buffer is defined by Equation 4.13. We can see a pseudo-code of this algorithm implemented in our kernel in Algorithm 7.

$$activation\_channelwise = Max(N_{ix}) \cdot Max(N_{iy}) \cdot P_{of} \cdot sizeof(DataType) \quad (4.13)$$

4.2. Theoretical Model

The aim of the thesis is to get latency benefits for FPGA inference by Channel Pruning using Genetic Algorithm search. We are using a framework (in section 4.3.3) to determine a set of compression ratios for each layer of DeepLab. As a requirement, a hardware model of our custom FPGA for DeepLab is required which is used to simulate the real hardware latency. This estimated latency metric by newly plugged in hardware model is used to guide the Genetic Algorithm to perform a latency based pruning search.

4.2.1. Design Variables of Hardware Model

The hardware model is able to predict the latency if the specification of each layer to be executed is provided. DeepLab consists of variety of different layers including Convolution (standard and dilated), Average Pooling and Upsampling. Naturally, the calculation to estimate latency for any layer varies largely. Please note that,

Algorithm 7: Pseudo-code for Upsampling OpenCL Kernel

```
for out_z=0; out_z < ⌈N_if / P_of⌉; out_z++ do
    for lane_div_vec=0; lane_div_vec < P_of / P_if; lane_div_vec++ do
        for i=0; i < N_ix; i++ do
            for j=0; i < N_iy; j++ do
                Read a vector Input vector from DRAM
                for ll=0; ll < P_if; ll++ do
                    In local buffer store entire activation for P_if channels
                end
            end
        end
    end
    for i=0; i < N_ox; i++ do
        for j=0; i < N_oy; j++ do
            for ll=0; ll < P_if; ll++ do
                Calculate coordinates of 4 pixels for each output pixel
                Calculate the weighted average value of output pixel
                Converted to float-point values and scaling factors multiplied
                Convert back to fixed point notation
            end
            Write into the channel to MemWrite
        end
    end
end
```

for the latency calculation of convolution layers, all the related functions like Max-Pooling, ReLU, Batch Normalization are already covered as they are dominated by convolution latency (compute-bound or memory-bandwidth bound). Here are the layer parameters given as input to the hardware model:

- Layer Type (Comvolution/Fused Convolution/Average Pooling, Bilinear Upsampling)
- Dilated Convolution (True/False)
- spatial dimension of input feature map (N_{ix} and N_{iy})
- spatial dimension of output feature map (N_{ox} and N_{oy})
- spatial dimension of weight kernels (N_{kx} and N_{ky})
- Number of input channels (N_{if})

- Number of output channels (N_{of})

Except these above parameters, we have some design choices and specification based constant parameters inside the FPGA based accelerator. This includes:

- Unrolling factor along input channel dimension(P_{if})
- Unrolling factor along output channel dimension(P_{of})
- Unrolling factor along x dimension of weight kernel (P_{kx})
- Precision of operands inside the hardware in Bytes (2 for 16 bit quantization)
- peak frequency of the FPGA (in Hz)
- DDR memory bandwidth (in B/sec.)

4.2.2. Convolution Latency Prediction

Compute Bound Latency Calculation

For strided convolution we reduced the number of DRAM access by using loop tiling technique. In Section 2.3.3, data re-use by tiling is explained where tiling a larger volume of input feature map helps to make the process compute-bound rather than restricted by global memory bandwidth. Due to the high computational demand of convolutional layer, tiling more input features does not improve latency and in this case the latency calculation is compute-bounded.

For MAC operations, the number of additions performed is almost equal to the number of multiplications. For ease of calculation total number of effective operations in a strided convolution layer in our custom accelerator having unrolling factors P_{if}, P_{of} and P_{kx} is given by Equation 4.14:

$$\#Effective\_Ops = 2 \cdot (N_{ox} \cdot N_{oy} \cdot \left\lceil \frac{N_{of}}{P_{of}} \right\rceil \cdot P_{of}) \cdot (N_{ky} \cdot \left\lceil \frac{N_{kx}}{P_{kx}} \right\rceil \cdot P_{kx} \cdot \left\lceil \frac{N_{if}}{P_{if}} \right\rceil \cdot P_{if}) \quad (4.14)$$

As explained in section 4.1.2, one DSP in the Arria 10 GX FPGA is capable of doing two 16-bit multiplications and two accumulations [72]. So the number of operations paralelly executed and the number of DSP blocks required for convolution operation are given by Equation 4.15:

$$\#Ops\_parallel = 2 \cdot (P_{if} \cdot P_{of} \cdot P_{kx}) \qquad \#DSPs = \frac{1}{2} \cdot (P_{if} \cdot P_{of} \cdot P_{kx}) \quad (4.15)$$

To get the number of cycles required for a convolution, we need to divide the number of effective operations (4.14) by the number of operations parallely executing (4.15) on available DSPs. The total number of cycles taken and the total latency in seconds for execution this layer is given in Equation 4.16:

$$\#Cycles = N_{ox} \cdot N_{oy} \cdot \left\lceil \frac{N_{of}}{P_{of}} \right\rceil \cdot \left\lceil \frac{N_{if}}{P_{if}} \right\rceil \cdot \left\lceil \frac{N_{kx}}{P_{kx}} \right\rceil \cdot N_{ky} \qquad Latency = \frac{\#Cycles}{Frequency} \quad (4.16)$$

Memory Bound Latency Calculation

For the convolution layers with dilation, it was not possible to use tiling technique in our current accelerator implementation. This is because of irregular re-use pattern of activations in tile buffer depending on the dilation factor. A special dataflow must be adopted to support tiling for dilated layers along with stride. So the latency calculation is memory bandwidth bound rather than being compute-bound. This estimation of latency consists of three different latency:

1. **Initialization Latency:** Time taken to load activation and weights from the memory. These two latencies are individually computed using Equations 4.17 and 4.18:

$$Latency\_in = \frac{(\left\lceil \frac{N_{kx}}{P_{kx}} \right\rceil \cdot P_{kx} \cdot N_{ky} \cdot \left\lceil \frac{N_{if}}{P_{if}} \right\rceil \cdot P_{if}) \cdot sizeof(DTYPE)}{MemoryBandwidth} \quad (4.17)$$

$$Latency\_weight = \frac{(\left\lceil \frac{N_{kx}}{P_{kx}} \right\rceil \cdot P_{kx} \cdot N_{ky} \cdot \left\lceil \frac{N_{if}}{P_{if}} \right\rceil \cdot P_{if} \cdot P_{of}) \cdot sizeof(DTYPE)}{MemoryBandwidth} \quad (4.18)$$

2. **Latency for Convolution:** This part of latency is responsible for the actual convolution operation by taking data from the SRAM. This is exactly similar to the latency calculation previously shown in Equation 4.14 - 4.16.

3. **Latency for Broken Pipeline:** This latency only comes into existence when the loop bound for DRAM is greater than the loop bound of SRAM. In that case the data pipeline breaks and data is shifted partially from DRAM to SRAM.

4.2.3. Latency for Average Pool and Bilinear Upsampling

We have 1 average pooling and 2 bilinear upsampling in DeepLabV3+ model. These special layers have a bit irregular type of computations than vanilla convolution layer. So to predict a near accurate latency value of each of these specific layers we resorted to unique linear regression functions or step functions on the experimental data.

Only for these layers, we experimented with different number of output channels (N_{of}) and got respective latency in ms. The line or step function that fits to these points are given in Appedix A.2. Below in Equation 4.19 we can see an example of step function equation to predict the latency of an Upsampling layer ($60 \times 60 \longrightarrow 240 \times 240$) of DeepLab network. Here 28.4 ms. is the latency of upsampling layer having upto P_{of} channels, and the value increases in steps of P_{of}.

$$Latency\_ms = 28.4 \cdot (\left\lfloor \frac{N_{if}}{P_{of} + 1} \right\rfloor) + 28.4 \tag{4.19}$$

4.3. Automated Channel Pruning for Custom Accelerator

4.3.1. Genetic Algorithm

Genetic algorithms (GA) are a class of evolutionary algorithms inspired from the process of natural selection. In natural selection, species that can adapt well to changes in the environment, survive and go to the next generation, a concept also termed as 'survival of the fittest'. Genetic algorithms simulate the process of natural selection, where the fitness of the individuals is defined by our desired optimization criteria.

At first, the population is initialized with a set of individuals. The fitness of each individual in the population is evaluated based on some fitness function. Offsprings are generated from the existing population through the process of variation. The process of variation involves $Crossover$ and $Mutation$. The fitness of the generated offsprings is evaluated. The fittest individuals from the parents as well as the children are identified through $Selection$ to go onto the next generation. This process

is continued till a maximum fitness value is reached or for a given number of generations. The process of *Crossover*, *Mutation* and *Selection* are elaborately discussed in Appendix A.3.

4.3.2. Choice of Genetic Algorithm: NSGA-II

Various variants of Genetic Algorithms are available in literature. For our purpose, we use the Non-dominated Sorting Genetic Algorithm II or NSGA II [73]. The pruning problem at hand demands solving a multi-criteria optimization problem with two opposing objectives. We aim at finding the layerwise sparsity ratios such that the computation effort in terms of number of operations or hardware estimate like latency is minimized. At the same time, we want to ensure that this does not affect the mIOU of Segmentation. However, aggressive pruning leads to a drop in mIOU, whereas, higher mIOU is costly in terms of hardware. Hence, there is no unique best solution but a trade-off between mIOU and latency has to be made. NSGA provides a set of pareto-optimal solutions that balances mIOU and hardware estimate.

This problem can also be solved using Simple Genetic Algorithm (SGA) or traditional Reinforcement Learning based search agents. However, in this case, a reward or fitness function needs to be formulated, that performs the trade-off between mIOU and hardware cost and provides a unique solution. The success of these methods is heavily reliant on the formulation of the reward function. NSGA, on the other hand, provides the flexibility to choose among a number of pareto-optimal solutions based on hardware constraints. Hence, NSGA is used for the pruning search problem.

4.3.3. Channel Pruning Framework using NSGA-II

This work focuses on structured pruning, more specifically channel pruning, since the advantages of channel pruning can be directly obtained in hardware. The NSGA algorithm is configured such that it maximizes the mIOU and simultaneously minimizes the hardware estimates. We conduct experiments using two hardware metrics:

- MAC operations: Initially experiments are conducted using number of operations, which is a proxy metric giving an indication of the required computational effort.
- Hardware Model Latency: NSGA search is also conducted using the latency values predicted by the theoretical model of our hardware accelerator, described in section 4.2.

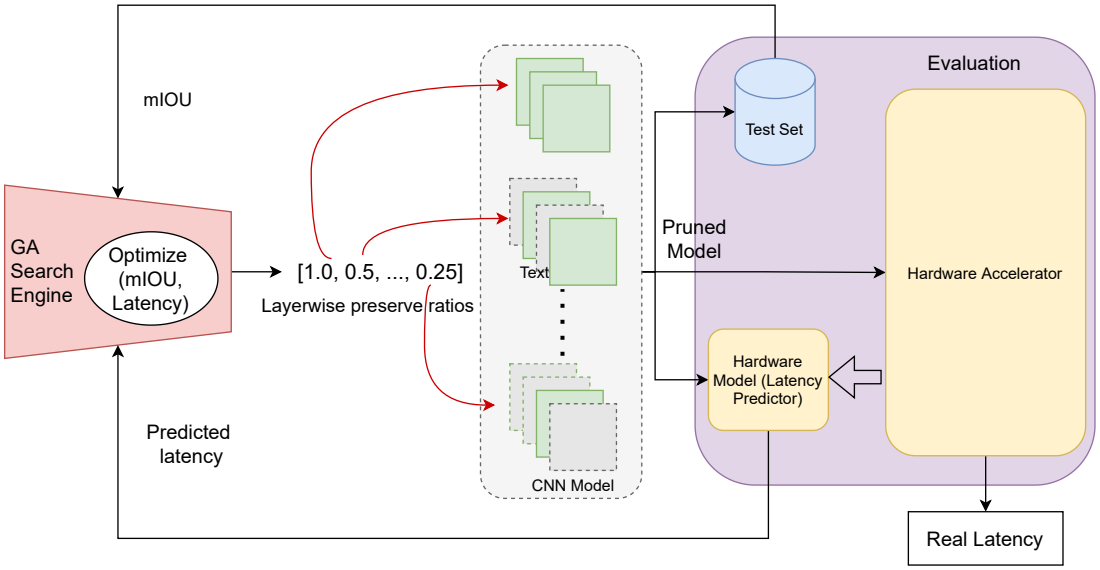

Figure 4.5.: GA Pruning Framework

An overview of the pruning framework is provided in figure 4.5. The GA search engine initially starts with a population of given size. Each individual in the population is a neural network, with randomly selected pruning ratios of its layers. These networks are evaluated to obtain their mIOU and hardware estimates. The mIOU values are obtained by evaluating the pruned model on the test set. If number of operations is optimized in GA search, the number of operations is calculated using layer-dimensions of the pruned model. For latency based GA search, latency values are calculated using the hardware model discussed in section 4.2. The mIOU and hardware estimate values are fed back to the GA, based on which the GA assesses the fitness of the individuals. The individuals in the population then undergo variation (cross-over and mutation) producing offsprings. The offspring are also evaluated as described above. Based on their fitness, a set of individuals are then selected from the parents and offsprings to go onto the next generation. This process is continued for a predefined number of generations. The algorithm 8 gives a walkthrough of the GA search procedure. After evaluation, the NSGA algorithm finds a set of non-dominated or pareto-optimal solutions. Pareto-optimal solutions are a set of solutions beyond which none of the objectives can be improved without sacrificing at least one of the other objectives. The pareto-optimal solutions obtained using NSGA have been illustrated in section 5.2.1.

Algorithm 8: Channel Pruning using NSGA-II.

Input : Initial population size $init\_sz$, Running population size mu, Number of layers in the network n, Crossover probability p_c, Mutation probability p_m, Lower bound and upper bound of compression for each layer: $lbound$ and $ubound$

Output : Pareto-optimal solutions balancing Accuracy and Hardware Estimates

1. Define individual: $[random\_uniform(lbound, ubound)$ for i in range$(n)]$
2. Initialize population P with $init\_sz$ individuals
3. $[Acc\_list, Estimate\_list] = Evaluate(P)$

for *index, ind in enumerate(P)* **do**
  $ind.fitness = [Acc\_list[index], Estimate\_list[index]]$
end

4. Update Pareto solutions

for *gen in range(1, max_gen)* **do**
  Offspring $O \leftarrow Variation\ (P, p_c, p_m)$
  $[Acc\_list, Estimate\_list] = Evaluate(O)$
  for *index, ind in enumerate(O)* **do**
    $ind.fitness = [Acc\_list[index], Estimate\_list[index]]$
  end
  $P \leftarrow select\_Tournament\ (P + O, mu, tourn\_size = 5)$
  Update Pareto solutions
end

4.3.4. Hardware Heuristics

Residual Network and DeepLab specific Hardware Heuristics

Due to presence of identity and shortcut connections in the residual layers, some feature maps must have the same dimension. For channel pruning, we need to ensure that the output channels (N_{of}) of the preceding layer or the input channels of the next layer, for these connections, are equal. This is necessary to facilitate element-wise addition. In Figure 4.6, these constraints are marked and the layers which should have same number of input channels are color-coded. It is interesting to notice that, for Conv2_1:1, Conv2_2:1, Conv3_1:short, Conv3_1:1 and Decoder1, it forms a chain, and every layer in the chain has same number of input channels.

Introducing Deeplab specific Compression Constraints in Pruning Search

The constraints on the number of input channels in feature maps also impacts the layerwise compression ratios, since the pruned model must also satisfy these constraints, when deployed in hardware. The compression ratios for the layers are

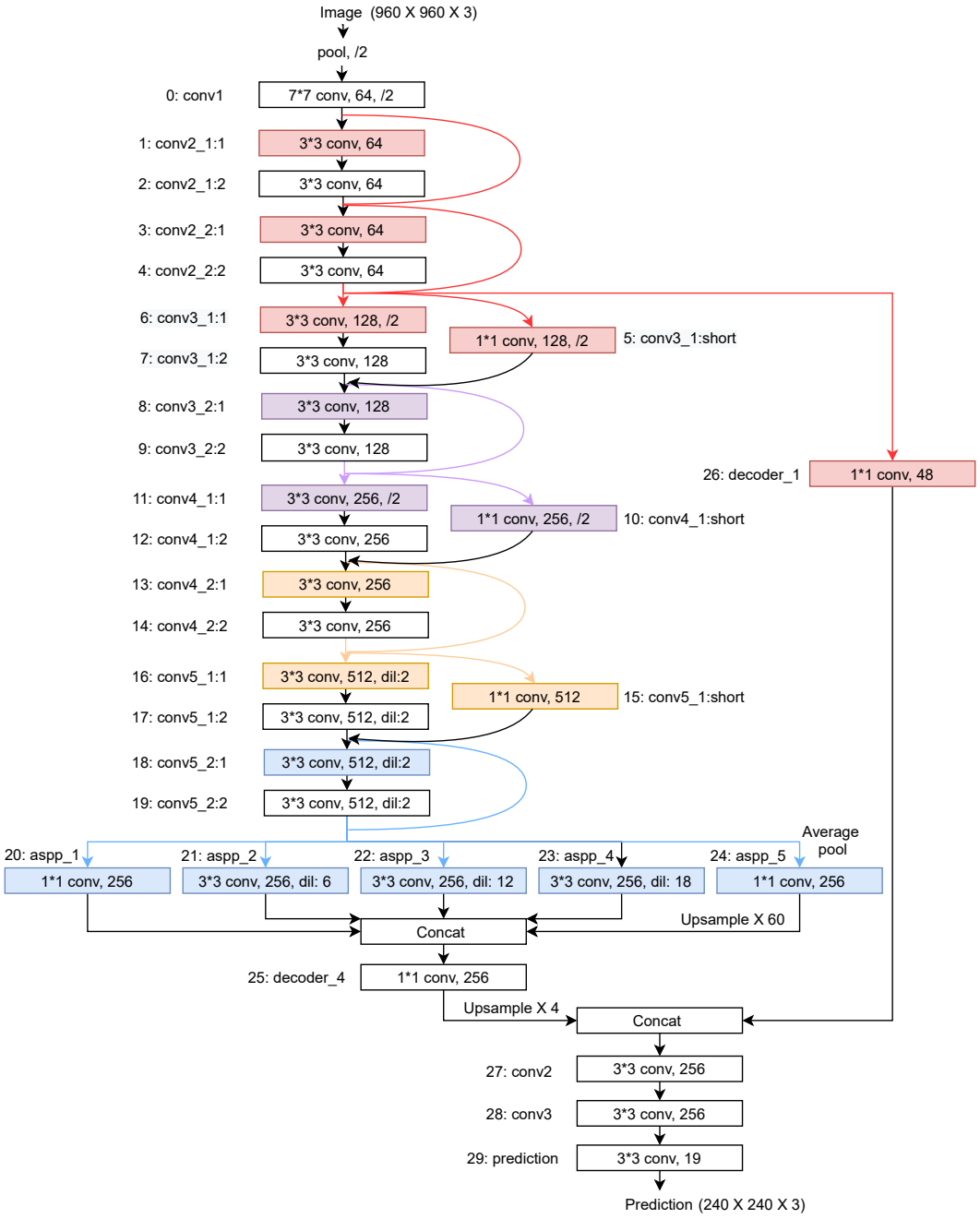

Figure 4.6.: Hardware Heuristics detected for DeepLabV3+ network architecture to guide the Genetic Algorithm search

usually chosen independently. However, if the constraints on compression ratios are not introduced during GA search, we need to consider an upper bound of the predicted channels for color-coded channel sets. This results in an increase in estimated hardware metrics. We illustrate in section 5.2.1 that imposing compression constraints during GA search yields better hardware estimates.

HW Model Latency based Pruning Search

The pruning search can be conducted using proxy metrics like number of parameters or operations. However, a decrease in number of operations does not necessarily mean a decrease in real latency, as illustrated in section 5.2.3. The actual hardware latency depends on various hardware parameters and is specific to the implementation of different layers in the hardware accelerator. Hence, an accelerator specific hardware model emulating the real latency is beneficial for guiding the search algorithm. In section 5.2.3, we illustrate that using a hardware model, we are able to obtain extremely low values of latency on our custom accelerator. Also, we achieve a much better trade-off between mIOU and latency.

5. Experimental Results

This chapter is based on the results of the experiments performed with the custom accelerator described in the previous Chapter 4 implemented on Arria 10 GX FPGA[15]. The baseline implementation results of DeepLabV3+ [14] network for semantic image segmentation on FPGA is described first in section 5.1. This covers experimental setup, special layers requirements in hardware, design space exploration, speedup due to tiling and various other metrics for different configurations of the accelerator. Following that, channel pruning benefits are demonstrated in Section 5.2. The aspects of ops based channel pruning and reduction in layer-wise latency have been considered in section 5.2.1. At the end, section 5.2.3 tells about hardware model of the accelerator to facilitate latency based channel pruning of DeepLabV3+ and shows improvement with respect to number of ops based channel pruning.

5.1. Image Segmentation using DeepLab V3+ on FPGA

5.1.1. Experimental Setup

Hardware

Different configurations of the accelerator were synthesized using Intel® FPGA SDK for OpenCL™ Offline Compiler [25] for Intel Arria 10 GX 1150 FPGA [15]. The synthesis machine is supposed to have decent specifications for memory intensive applications in the OpenCL Offline compiler. A machine with Intel® Xeon® CPU and 128 GB memory was used for synthesis. The host side application was running on Intel® Core i7-3770K CPU with which the FPGA was connected and the compiled OpenCL kernels were running on the FPGA.

The unrolling factors along the input channel (P_{if}), output channel (P_{of}), kernel (P_{kx}) dimensions and precision of operands are configurable before the synthesis start. Also, the size of the buffers required by the OpenCL kernels are configured beforehand (Section 4.1.3) by computing the requirement of most memory resource intensive layer of the segmentation model considered. At runtime, several items can be configured in the host side application such as the network architecture, respective scaling factors (SF_i, SF_w and SF_o) of layers and any post-processing of data coming out of FPGA. Alongside with a separate configuration file, the programmable aocx

file (synthesized), validation references and tiling factors (T_{ox}, T_{oy}) can be customized which can improve latency of the network.

Segmentation Model

In the experiments, to get real-time semantic image segmentation we used Deeplab [58] model by Google which was introduced in 2016. It controls filter's field-of-view using a parameter *dilation rate* which is typically called *Atrous Convolution*. Since then several improvements have been proposed. In DeepLab V2 [74] to tackle the problem of segmenting objects in multiple scales, a pyramid pooling block using Atrous Convolution was introduced. In V3 [44], image level feature pooling which encodes global context was incorporated and in the latest version V3+ [14] a decoder module was added to refine the segmentation result. For our accelerator the network was implemented using 16 bit precision of weights and activations. The DeepLab network has ResNet [34] as a backbone and residual block are frequently duplicated to be used in cascade or in pyramid pooling manner. In our all experiments we used DeepLabV3+ with ResNet18, which includes a Encoder-Decoder type architecture and all the previous DeepLab features. The shapes and desciption of all layers of the network is available in section 2.4 of Chapter 2.

Dataset

With the DeepLab network, we used Cityscapes [59] as the dataset. It contains high-quality pixel-level annotations of 5000 images for semantic understanding urban street scenes. 960×960 images with 3 channels images were used as a input to the FPGA which were preprocessed from 2048×1024 sized raw images. Total 19 classes were used for annotating segmentation output and the resulting mean Intersection-Over-Union was 67.27%.

5.1.2. Memory Requirement for Dilation

An integral part of any iteration of DeepLab network is Dilated Convolution or Atrous Convolution. An additional hyperparameter (dilation rate) was introduced to inflate the kernel. This is done by inserting some additional zeroes in between the weight kernel elements. As a result we get an enlarged kernel in spatial dimension, which was described in Chapter 2. In the FPGA based accelerator, it is required to store a block of weights in local SRAM for computing convolution of each of the output activations of a channel. Naturally, the memory requirement increases with increasing kernel size in x and y dimension. For higher dilation rates the weight buffer size explodes which results an enormous requirement of on-chip memory to implement that particular layer in the hardware. For example, with dilation rate 18 for a 3×3 kernel, it inflates to size of 37×37. With a input channel size of 512 it requires around 22.5 MB in the on-chip memory (calulated with precision

16, (P_{if})=16, (P_{of})=16, (P_{kx})=1). In the Arria 10 FPGA there are two types of memory blocks: M20K blocks and memory logic array blocks (MLABs). In our specific FPGA model, we have around 7.2MB memory available comprising of the two types of memories. Naturally in the pyramid pooling, layers with dilation 12 and 18 requires way more higher memory than they are available in the FPGA. In Figure 5.1, we can see the memory requirements of the extended dilation layer weights and the same in our implementation. Please note that the Y-axis is in log-scale and the horizontal line signifies the available on-board memory. A detailed weight memory requirement of all the layers of DeepLabV3+ network can be found in Appendix A.1.

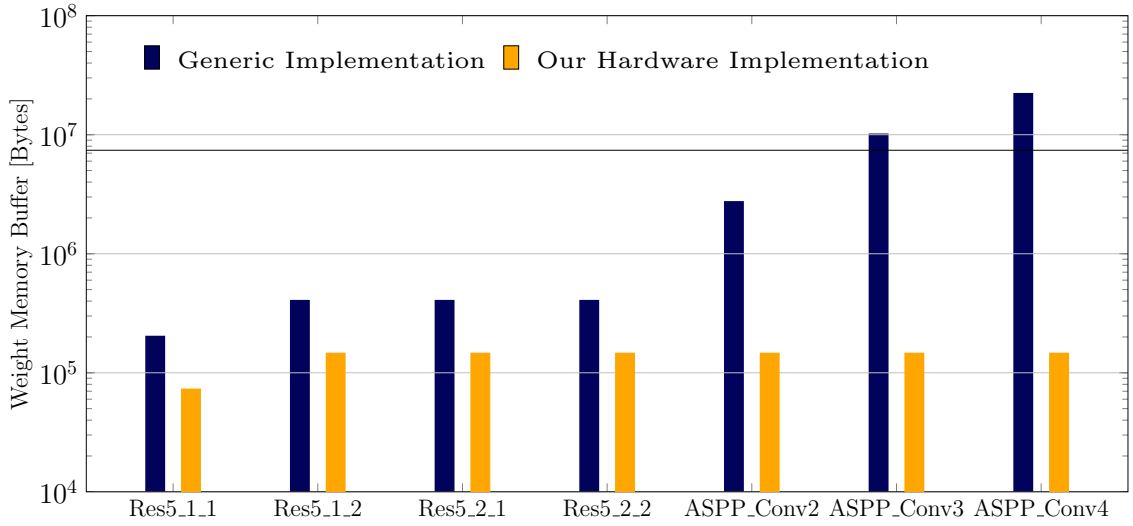

Figure 5.1.: Weight Buffer comparison in generic and our hardware implementation of Dilated Convolution

5.1.3. Design Space Exploration

Loop Unrolling Based Configurations

In this section, resource utilization of different versions of the DeepLabV3+ accelerator for deployment in Arria10 GX FPGA has been considered. The High-Level Synthesis (HLS) was done using AOC version 19.4.0 and Quartus 19.4.0 Build 64. The reference accelerator model is being taken from PipeCNN [48], where unrolling along the input and output channel dimensions (P_{if} and P_{of}) were introduced. From this research practice report [40], unrolling along the horizontal axis of the kernel(P_{kx}) was also possible.

In most semantic segmentation networks there are other special type of layers involving operations other than vanilla convolution. In hardware implementation of DeepLabV3+, the dilated convolution feature has already been introduced in the

convolutional computational unit as explained in section 4.1.2. Along with dilation there are special OpenCl kernels required which are responsible for average-pooling and bilinear upsampling the input activations. Theoretically concatenation feature is also necessary which we tactically avoided by exploiting the fuse layer functionality already implemented for residual layers in ResNet architecture.

Introduction of these new kernels affects the FPGA resource utilization as they consume additional logic, memory and DSP. These average-pooling and bilinear upsampling does not need any weights to operate. Naturally, scaling up OpenCL kernels in the input and output channel dimensions were introduced. In Table 5.1, we can see the resource utilization of different configurations and the aim of the table is to point the variation of the resource demands.

Unroll config P_{if}, P_{of}, P_{kx}	**Logic Utilization**	**Registers**	**DSP blocks**	**BRAM M20K**	**Frequency MHz**
16, 16, 1	41%	327698(19%)	394(26%)	960(35%)	208.33
16, 32, 1	58%	456043 (27%)	690(46%)	1478(64%)	189.81
16, 64, 1	68%	582430 (34%)	617 (41%)	2175 (80%)	161.46
16, 16, 4	51%	415308 (24%)	778 (51%)	1327 (49%)	190.97
16, 32, 4	72%	588693 (34%)	1362 (90%)	2025 (75%)	148.44
Max Available Resource	100%	1708800	1518	2713	800

Table 5.1.: Resource Utilization of DeepLabV3+ Accelerator for different configuration of Unrolling factors

In Arria10 GX 1150 FPGA, maximum available resources are mentioned in the last row. This FPGA contains two types of memory blocks: M20Ks (20Kb/each block) which are suitable for larger memory arrays and memory logic array blocks (MLABs, 640 bit each) which are usually used for shift register implementation in DSP applications. The Flip-Flops are Dedicated Logic Registers and a fixed amount of maximum DSP blocks (1518) are available on board. As a baseline implementation, an accelerator with configuration P_{if}=16, P_{of}=16 and P_{kx}=1 was synthesized. Any significant stall in the whole acclerator dataflow pipeline was identified with the Intel FPGA Dynamic Profiler for OpenCL [71]. The stalls were tackled with various preprocessing diretives (*pragma*) which forces the offline compiler to do the implementation in a specific way. For example, unrolling the primary loop in the OpenCL kernel responsible for Batch Normalization operation helps to mitigate stall in the pipeline. In subsequent implementations, unrolling in the output channel dimension, i.e. $P_{of} = 32$ and 64 were considered which naturally resulted in higher logic and RAM requirement. Unfortunately with $P_{of} = 64$, the logic requirement was >100%. So different measures were taken to minimize the logic utilization. For

example, offline compiler was forced to implement pipelined Load-Store Unit (LSU) instead of Burst-Coalesced LSU and fusing scaling factor multiplication through BatchNorm kernel was implemented. Even then, unrolling the primary loop fully in BatchNorm kernel was not synthesizable. As expected, latency performance was inferior because of the stalls present in the pipeline (for not being able to unroll the BatchNorm kernel). Next, the accelerator was scaled in the P_{kx} direction which has much more usage of on-board DSPs. But as lesser unrolling factor was considered in the P_{of} direction, the logic and memory requirement was reduced from the previous iterations. The final baseline considered was both scaled up in P_{of} and P_{kx} direction and utilizing around 90% of on-board DSPs. Here also several techniques were used at the kernel level to keep logic requirement minimal. We got the best latency performance from this configuration which is described in section 5.1.3.

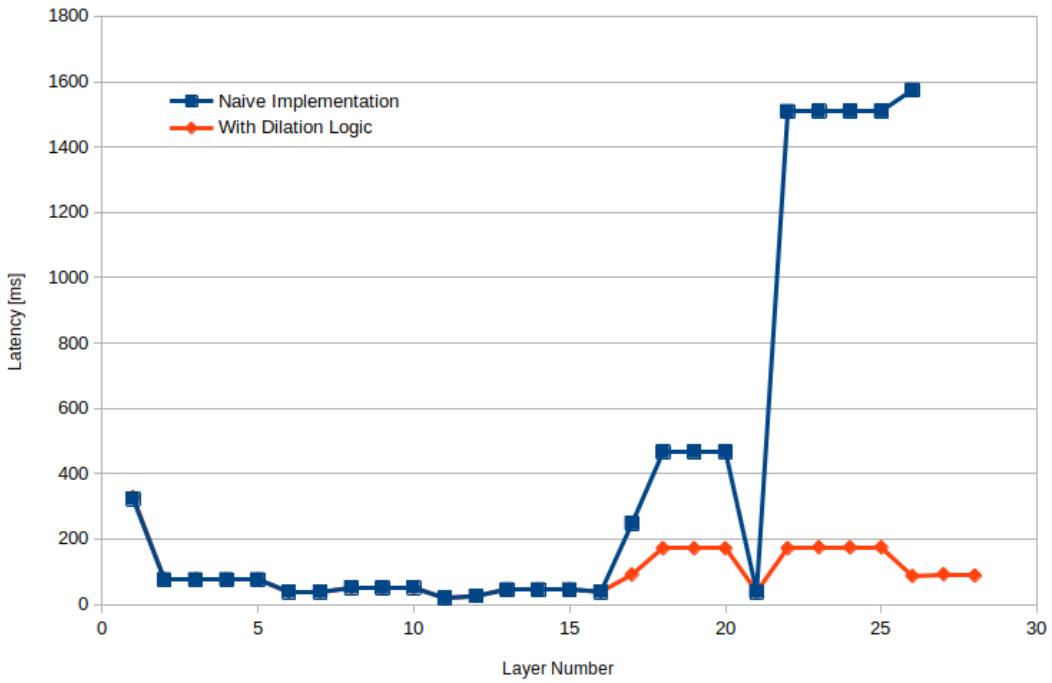

Figure 5.2.: Measured Latency of the different Convolution layers of DeepLab with and without Dilation feature in accelerator configuration $P_{if} = 16$, $P_{of} = 16$ and $P_{kx} = 1$

Dilation layer speedup with respect to Naïve Implementation

In this section, the speedup of execution of dilated layers will be briefly discussed in comparison with general Convolution implementation. If dilated kernels are computed with zeros in-between the effective weights, the actual number of operations

are very high. In Figure 5.2, this comparison of execution times has been plotted for the accelerator configuration $P_{if} = 16$, $P_{of} = 16$ and $P_{kx} = 1$. Please note that, for layer 27 and 28, i.e. ASPP layers 4 and 5, where dilation factors are respectively 12 and 18, the naïve implementation is not synthesizable for higher memory requirement(section 5.1.2). As we are picking selected weights directly from the DRAM depending on the dilation ratio of the layers we get a speedup of 2.7×, 8.8×, 18.3× for dilation factor 2, 4 and 6 respectively.

Baseline results of Segmentation

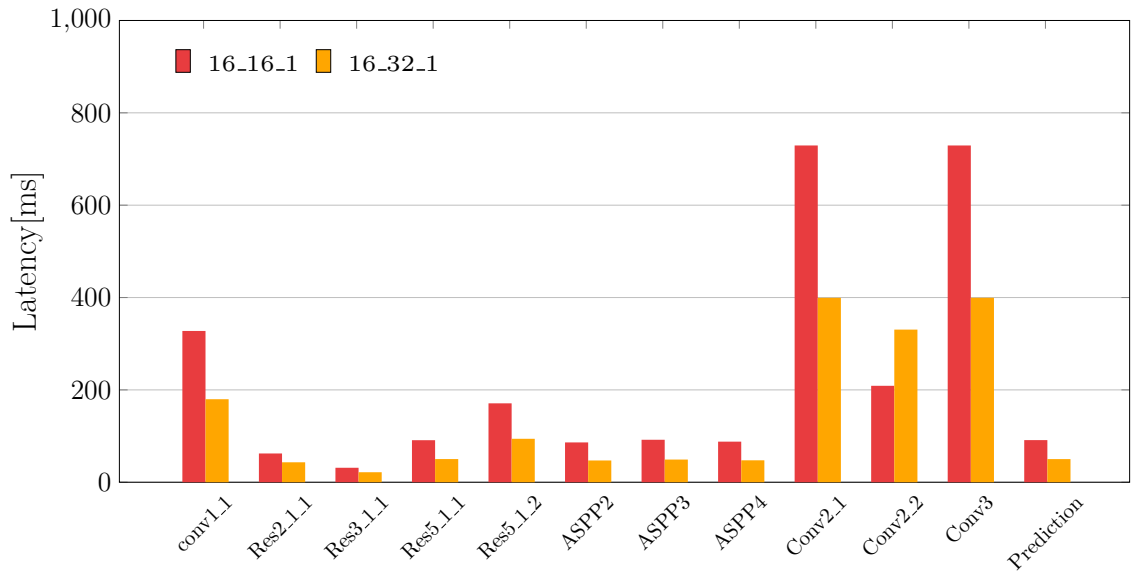

Figure 5.3.: Latency comparison of computation intensive layers of DeepLab for accelerator configuration ($P_{if} \times P_{of} \times P_{kx} = 16 \times 16 \times 1$), and scaled up accelerator along output channel with configuration ($P_{if} \times P_{of} \times P_{kx} = 16 \times 32 \times 1$) having no tiling implemented.

As mentioned earlier, we maximized the use of computational resources on the FPGA and by unrolling the loops in 3 directions. The motivation is to improve inference time by increasing the DSP utilization. At the end we got an accelerator with $16 \times 32 \times 4$ configuration which had most DSPs utilized while synthesis(around 90%). However, while measuring inference latency, we have to remember that DeepLab has ResNet architecture as backbone which contains memory bounded residual blocks. Now step by step, let us see how the latency performance improves with subsequent loop unrollings. In Figure 5.3, we can see the measured execution times of the most computation intensive layers of DeepLabV3+ architecture (which we slightly modified for efficient hardware implementation) for the baseline $16 \times 16 \times 1$ and another P_{of} scaled $16 \times 32 \times 1$ accelerator. Please note that the layers with similar

number of operations and lower operations were not plotted to maintain readability. It is evident that we got execution time benefits in most layers. There are a few exceptions like Conv2_2 layer where the number of output channel is 48, which is not a multiple of 32. Naturally it does some extra computations resulting slightly higher latency than the $16 \times 16 \times 1$ configuration. Overall, for segmenting a single 960×960 cityscapes [59] image as input, the $16 \times 32 \times 1$ had a speedup of $1.38\times$ over the baseline accelerator configuration $16 \times 16 \times 1$.

What really matters for the inference performance is the percentage of useful computations happening in the all the DSPs, and this is termed as *DSP Efficiency*. To analyze further, the complexity (in #GOPs), DSP efficiency and actual throughput (in GOPS) of each layer are reported in Table A.3 (Appendix A.4) for 3 accelerator configurations. Although the $16 \times 32 \times 4$ has the most number of DSPs available for the computation, it does *not* result the highest throughput because of potential pipeline stalls and restricted memory bandwidths. Further improvements in this aspect have been demonstrated in the next section 5.1.4 via memory tiling .

In the $16 \times 16 \times 1$ accelerator we see a overall DSP efficiency of 93% and in most of the layers it has full DSP usage except the first and prediction layer. This is for the reason that the input activation has 3 input channels (not a multiple of 16) and prediction layer has a 19 numbers of output channels (also not a multiple of 16). On the other hand in the accelerator which is scaled in the P_{kx} direction has a lot more DSP in disposal. For $16 \times 32 \times 4$, it has potentially 8 times more DSPs blocks than the accelerator with unrolling configuration $16 \times 16 \times 1$. Still the layer dimensions does not always match with P_{kx} unrolling factor and produces lesser DSP efficiency. For example, for weight kernels with spatial dimension 3×3 and 1×1, this accelerator will always have DSP efficiency as $(3/4) = 75\%$ and $(1/4) = 25\%$ respectively at runtime. Overall this configuration has DSP efficiency of 68.23% but actual throughput is slightly better than $16 \times 16 \times 1$ configuration.

5.1.4. Latency improvement for Data Tiling

The DeepLab model is comparatively a complex architecture which contains a ResNet Network. Hence the time needed for transferring data from DRAM may be significant and comparable to convolutional computations. Earlier, in PipeCNN [75] only one tiling factor T_{ox} could be configured during the synthesis process and it is called as *static tiling*. In previous works [76] and [40], a new technique of *dynamic bi-directional tiling*, has been adapted where the tiling factors can be configured at inference runtime depending on the output spatial dimensions of individual layers. This two-directional (T_{ox} and T_{oy}) tiling technique is very effective with the convolutional layers which has low spatial dimension in output feature map (N_{ox}). Potentially the full feature map in the x direction can be tiled in the on-chip buffer and the remaining space could be used to load more features in the vertical direction, maximizing data reuse.

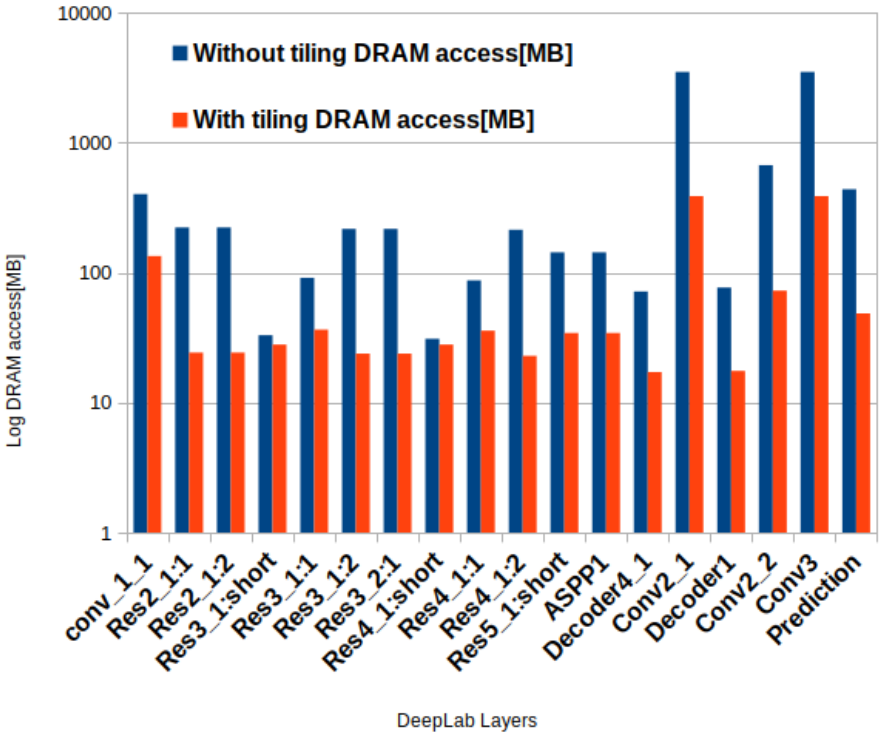

Figure 5.4.: Comparison of DRAM access in MegaBytes between no tiling and with tiling for the accelerator with configuration $P_{if} = 16$, $P_{of} = 32$ and $P_{kx} = 4$

In this section we analyze how the DRAM access rate and eventually latency drops for higher tiling factors, configured during runtime. Here we take our most scaled up version of accelerator designed for segmentation. In Appendix A.2, the DRAM access count along with the inference latency of each layer of DeepLabV3+ has been reported for untiled and 2D dynamically tiled accelerator with configuration $P_{if} = 16$, $P_{of} = 32$ and $P_{kx} = 4$. Tiling all the layers with same tiling factor is inefficient because in that case the tiling factor will be restricted by a layer with high input channel number (N_{if}). The size of the tiling buffer calculation (refer section 2.3.3) is dominated by input channel dimension(N_{if}). So another layer with less N_{if} might use a higher tiling factor to get lesser DRAM access count. In our accelerator we used a tile buffer size of 64×1024 bytes to store the sub-volume of input feature map ($T_{ix} \times T_{iy} \times N_{if}$) loaded from off-chip memory, so that we can try sufficiently high tiling factors for most layers. It is important to remember that in our current architecture, we are not able to tile the convolutional layers which have dilation value> 1. A more complex dataflow needs to be adapted to facilitate data reuse for dilated layers via tiling. Also the special layers like bilinear upsampling

and average pooling does not have opportunity to exploit tiling, and for maxpool layer tiling can only be exploited in the T_{ox} direction. By applying 2D tiling factors of T_{ox} and $T_{oy} = 10$ and 10 respectively to most conv layers, we got a 9× DRAM access benefit. Overall for the DeepLabV3+ architecture we got more than 2 times DRAM access benefits.

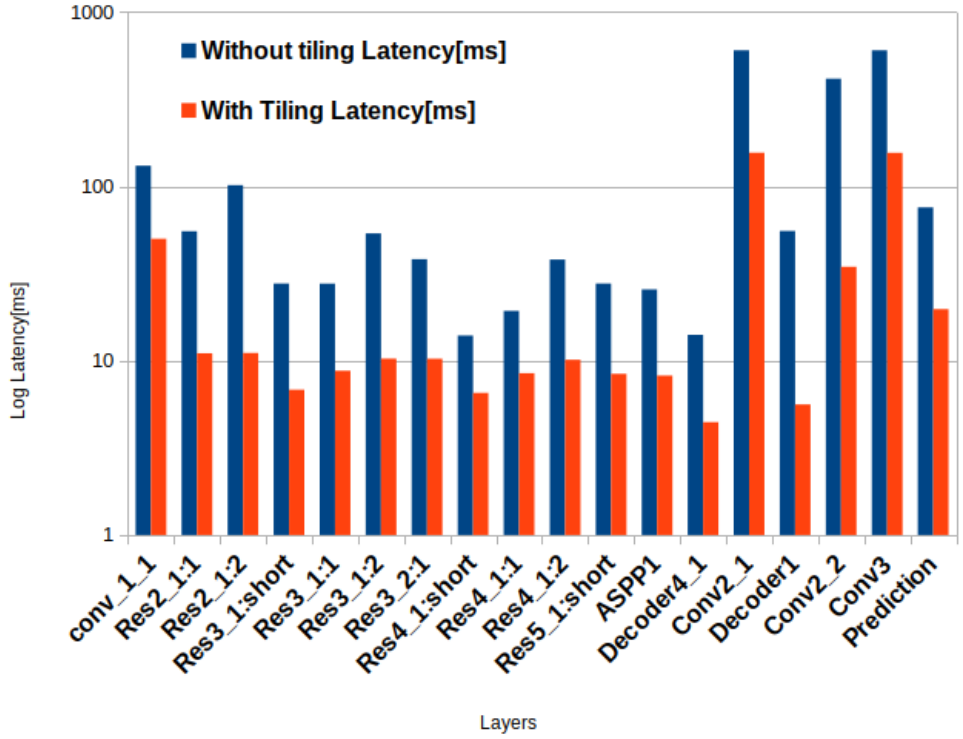

Figure 5.5.: Comparison of latency in ms. between no tiling and with tiling for the accelerator with configuration $P_{if} = 16$, $P_{of} = 32$ and $P_{kx} = 4$. Latency(ms.) in y-axis is in log-scale.

In Figure 5.4, the influence of 2D tiling on the DRAM access are clearly shown for multiple layers of DeepLab. Here also for better visualization the y axis is plotted as log of DRAM access in MB and only layers with noticeable reduction has been taken. It is interesting to note that, sometimes increasing the tiling factors do not influence the latency. At this point the pipeine is full and we are restricted by limited memory bandwidth. From Figure 5.5, we can see the latency reduction (due to tiling) in ms., of the same layers considered in the previous diagram and how the off-chip memory access count actually influences the overall latency. In total, we got a final latency of 1614.608ms (improvement of 2.3× with respect to not tiled version) in a tiled accelerator with configuration $16 \times 32 \times 4$. The overall throughput for this configuration is 182.806 GOPS. It is the best inference performance we got with

the baseline architectures, which is used for the experiments in following sections dealing with network compression. As a summary, in Table 5.2, we get latency comparison of accelerators with different unroll configurations, all exploiting data reuse by tiling.

P_{if}, P_{of}, P_{kx}	$16, 16, 1$	$16, 32, 1$	$16, 32, 4$
latency[ms]	3228.235	1928.854	1614.608

Table 5.2.: Tiled and Scaled Semantic Image Segmentation Accelerators

5.2. Channel Pruning for Segmentation

Channel pruning is performed on the Deeplab model with an objective to further reduce the model latency, when deployed in hardware. For this purpose, a pre-trained Deeplab model is considered. Genetic search is performed in order to obtain the layerwise pruning ratios, that provide the best trade-off between mIOU and hardware estimates. Finally, the pruned model is fine-tuned for 40 epochs to recover its mIOU value.

The NSGA algorithm implementation uses the DEAP framework [77]. The experiments are performed with an initial population size of 50 and a running population size of 25, after the first generation. The individuals are defined as a list of compression ratios of the layers of the CNN. We use *one-point* crossover and *replace* mutation with a cross-over probability p_c and mutation probability p_m of 1.0 and 0.4 respectively. The GA search is performed for 25 generations.

5.2.1. Ops-based Pruning without compression constraints

GA search is initially performed using number of operations as a hardware estimate, without considering any constraints required for deploying the model in real hardware. This returns a set of compression ratios for the layers, that are independent of one another. However, as explained in section 4.3.4, some feature maps must have the same number of channels. Therefore, the compression ratios of their corresponding layers must also be the same. Since this is not taken into account here, when deploying the model, we need to take an upper bound of the predicted number of channels for these layers. This leads to an increase in overall latency of the model, the **minimum possible latency achieved being 531ms**.

The figure 5.6, shows a plot of mIOU vs operations on the left and mIOU vs real hardware latency on the right. The grey dots in the left figure are the pareto solutions of each generation, with darker dots representing higher generations. The pareto solutions of the last generation are used to construct the pareto-front (displayed in red). The fine-tuned mIOU values for the respective latency reductions

are shown as green dots. The mIOU vs real latency (on the right), for these pareto-optimal solutions, further prove that a reduction in number of operations does not necessarily guarantee a reduction in network latency.

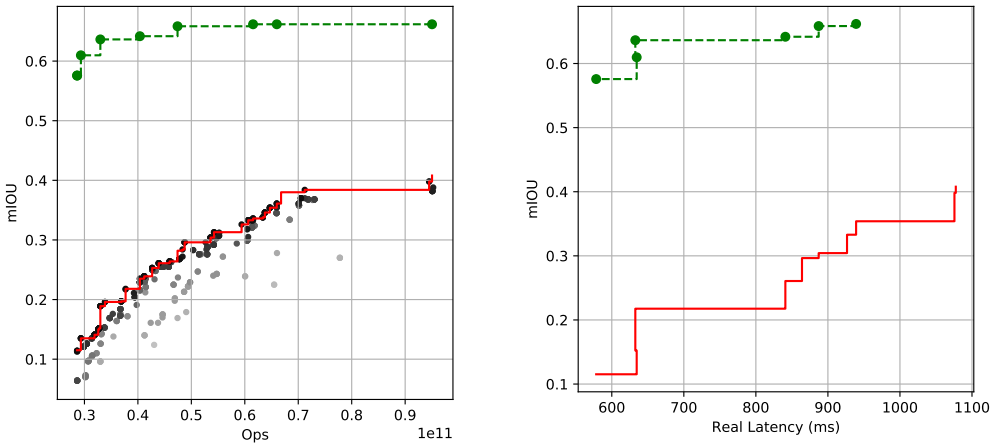

Figure 5.6.: NSGA Search using Ops without compression constraints

5.2.2. Ops-based Pruning with compression constraints

In the next step, GA search is performed using number of operations but taking into consideration the hardware constraints, described in section 4.3.4. The solutions obtain show better latency values, with the **minimum achievable latency being 349 ms**. Figure 5.7 shows the generation wise results for GA search on the left and mIOU vs real latency on the right, similar to figure 5.6. Even in this case, the pareto solutions obtained using operations do not follow the same trend for real hardware latency.

5.2.3. Hardware Model Latency guided Pruning with compression constraints

The failure of proxy metrics like operations to provide accurate estimates of real hardware metrics motivated us to develop the hardware model to better guide the GA search algorithm. Figure 5.8 illustrates the combined plot for all the three GA search scenarios. The hardware model latency values are shown on the left whereas real latency values are shown on the right. The graphs on the left and the right are identical in nature, indicating the accuracy of our hardware model. Using hardware model latency for GA search, we achieved a **minimum latency of 263 ms**, which is by far the best. The blue curve, which considers HW model latency

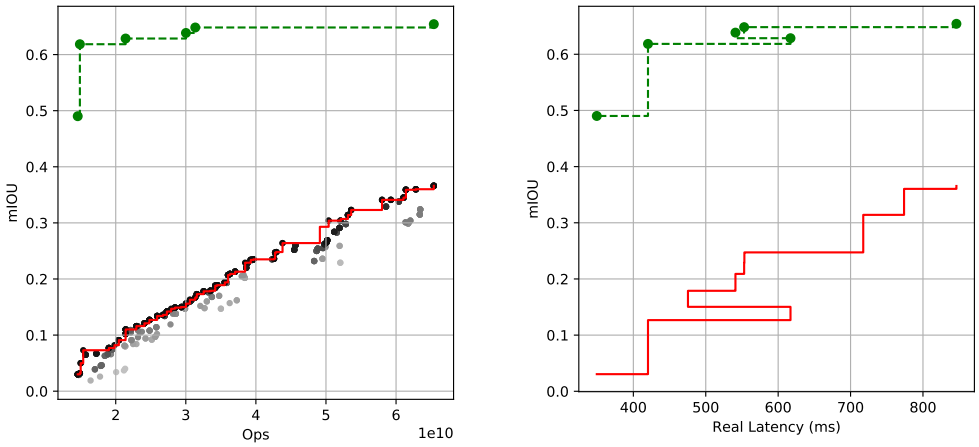

Figure 5.7.: NSGA Search using Ops with compression constraints

during search, also yields better trade-off between mIOU and latency. Considering the same mIOU of 0.63, the three methods achieved latency of 608 ms, 541 ms and 400 ms respectively, whereas the baseline latency obtained is 1614 ms. Thus, **for a degradation of 4% in mIOU, we are able to achieve a speedup of** $4\times$**. Also, incorporating the hardware model provides an improvement in latency by** $1.52\times$ **as compared to ops**.

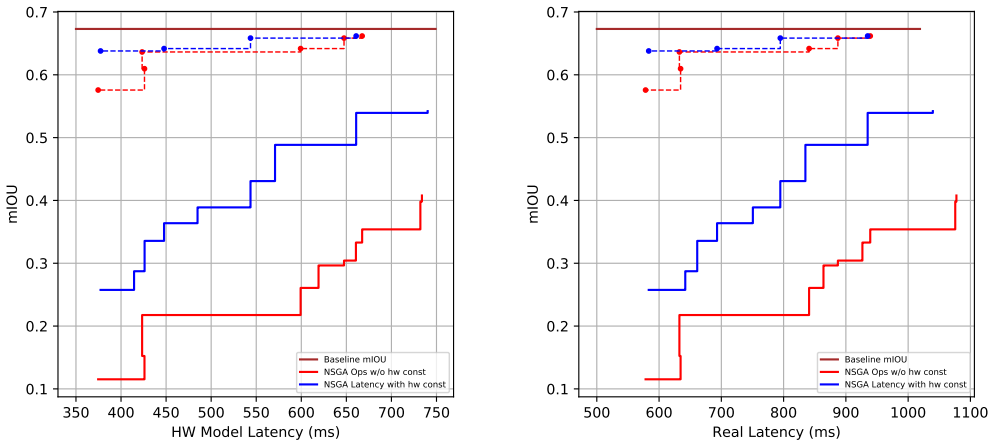

Figure 5.8.: NSGA Search using HW Model Latency with compression constraints

In the below Figure 5.9, a qualitative analysis of the hardware aware pruning framework for our custom accelerator has been demonstrated. We see 3 raw frames from CityScapes dataset and the respective ground truths (pixel-wise annotation).

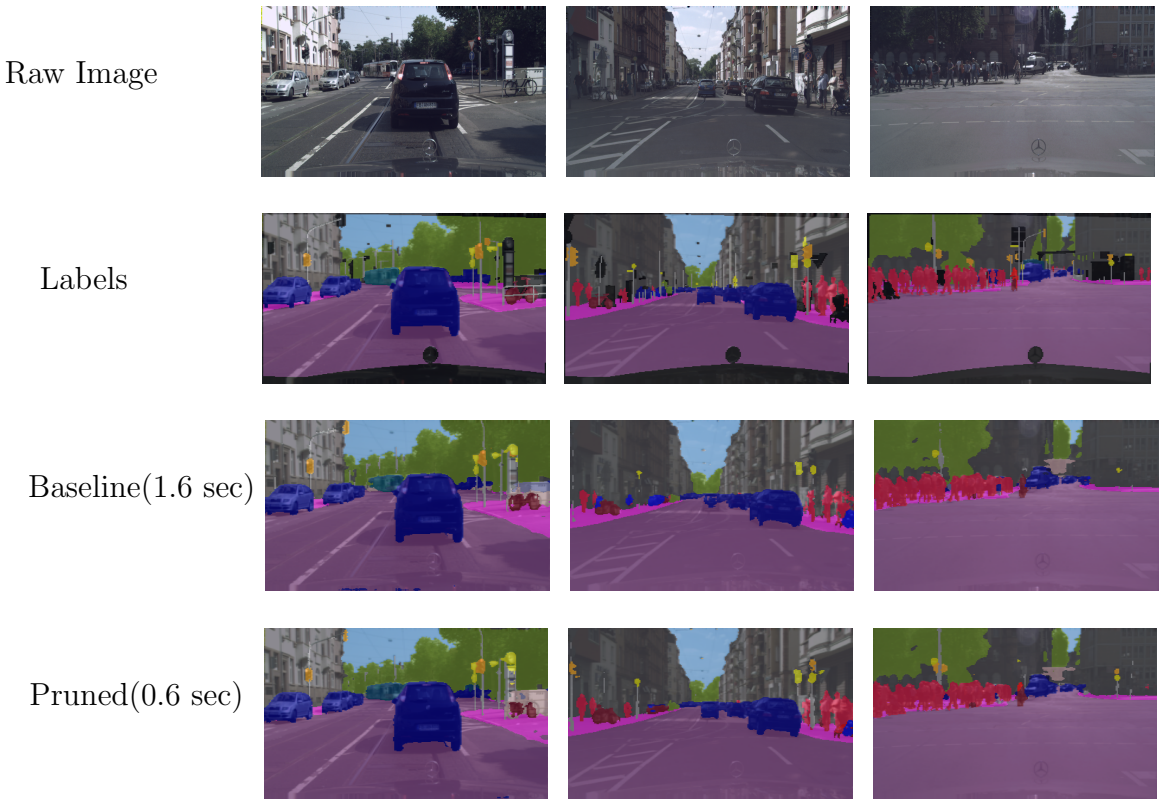

Figure 5.9.: Qualitative results for HW aware pruned models on different scenarios in the CityScapes dataset. Black regions are unlabeled in the original dataset.

We compared the segmented output of baseline implementation and another one using the pruned network (hardware model guided latency based GA search). With a mIOU drop of around 2%, we achieve a significant speedup of 2.7×. Visually, there are a few pixel differences noticeable, for example in the first image, less number of pixels are classified as 'bike' in pruned network compared to the original network.

6. Conclusion and Outlook

6.1. Conclusion

In this thesis, the problem of accelerating a state-of-the-art semantic segmentation network is successfully tackled by a FPGA based accelerator exploiting hardware-aware channel pruning technique. The entire system was designed with High-Level-Synthesis methodology which helped to cut down the development time and a number of design reiterations incorporating new architectural optimizations were possible. This architecture is fully pipelined and the computational units can be easily reorganized to accelerate a completely different network. Our platform provides a configurable architecture suitable for any DNN use case providing the maximum use of available resources.

Instead of using full precision floating point operations, this work leverages 16-bit fixed point representation for both weights and activations. A concept of scaling factor was introduced for handling this quantization problem. For further logic optimization, the operations for maintaining precision of the operands were fused for convolution and batch normalization. The fundamental architecture [48] was improved by exploiting three dimensional loop unrolling technique, that improved the latency and throughput. Furthermore, data tiling and double streaming-buffer were used to reduce number of off-chip memory access by re-using input feature map elements from a tile of data. Several DeepLabV3+ specific optimization measures were also taken care of including the implementation of novel dilated convolution. The computational unit, reading data from memory smartly selects the relevant input elements according to the dilation ratio of the specific layer. Alongside, dedicated computational units were developed to tackle bilinear upsampling and average pooling of input feature maps. Our baseline accelerator (before channel pruning) gives 183.293 GOPS throughput and takes 1.6 sec to process a single frame of 960×960 RGB CityScapes image.

At the end, we proposed a framework for automated channel pruning of segmentation algorithm on our custom hardware. We identified specific hardware heuristics that necessitates to have equal compression ratios for some specific layer of the underlying ResNet architecture. With this, also a hardware model was crafted that simulates the latency of the actual FPGA accelerator. These metrics and heuristics were introduced in the Genetic Algorithm based search giving a $4\times$ speedup over baseline model with a degradation of 4% mIOU accuracy. Also compared to proxy-metric (#operations) based GA search, this gave over $1.52\times$ speedup.

6.2. Future Work

During the implementation of this project several possible improvements were noted and can be incorporated in future implementations:

- **Tiling Dilated Convolution**: We use a tiling buffer in our architecture to store inputs for multiple consecutive convolutions. A data sharing possible within a tile and thus it reduces the number of DRAM access significantly. But in case of convolution layers with dilation, the data-reuse pattern is not straightforward. The rate of data reuse also reduces with increasing dilation ratio. For example, a layer with dilation ratio = 2 will have reuse benefit when $T_{ox} \geqslant 3$. A dedicated dataflow can be designed with a support to tile the dilation layers along with vanilla convolution.

- **Efficient Bilinear Upsampling**: In current implementation, the compute unit responsible for bilinear upsampling consumes a significant amount of time while inference. In DeepLabV3+, there are two bilinear upsamplings ($1 \times 1 \longrightarrow 60 \times 60$ and $60 \times 60 \longrightarrow 240 \times 240$) and this is responsible for around 15% of total latency. More efficient implementation and concepts like unpooling and deconvolution can be investigated to formulate a more efficient computational unit.

- **Smartly Detect Zero Channels**: We formulated a hardware-aware channel pruning technique which uses genetic algorithm for searching optimal compression ratios. Further modifications in the memory read computational unit can be done so that it can intelligently detect a zero channel coming from a previous layer, and subsequently skip the relevant operations. This can result an improvement on the latency performance for channel pruning in our accelerator.

A. Appendix

A.1. Experiments and Design Space Exploration

A.1.1. Weight Buffer Requirement in Memory

		Extended Kernel				Unextended Kernel		
Block	Layer	weight_w	weight_h	input_n	Buffer[B]	weight_w	weight_h	Buffer[B]
conv_1_1	1	7	7	3	4704	7	7	4704
Res2_1	2	3	3	64	18432	3	3	18432
Res2_1	3	3	3	64	18432	3	3	18432
Res2_2	4	3	3	64	18432	3	3	18432
Res2_2	5	3	3	64	18432	3	3	18432
Res3_1	6	1	1	64	2048	1	1	2048
Res3_1	7	3	3	64	18432	3	3	18432
Res3_1	8	3	3	128	36864	3	3	36864
Res3_2	9	3	3	128	36864	3	3	36864
Res3_2	10	3	3	128	36864	3	3	36864
Res4_1	11	1	1	128	4096	1	1	4096
Res4_1	12	3	3	128	36864	3	3	36864
Res4_1	13	3	3	256	73728	3	3	73728
Res4_2	14	3	3	256	73728	3	3	73728
Res4_2	15	3	3	256	73728	3	3	73728
Res5_1	16	1	1	256	8192	1	1	8192
Res5_1	17	5	5	256	204800	3	3	73728
Res5_1	18	5	5	512	409600	3	3	147456
Res5_2	19	5	5	512	409600	3	3	147456
Res5_2	20	5	5	512	409600	3	3	147456
ASPP1	21	1	1	512	16384	1	1	16384
ASPP2	22	13	13	512	2768896	3	3	147456
ASPP3	23	25	25	512	10240000	3	3	147456
ASPP4	24	37	37	512	22429696	3	3	147456
ASPP5	25	1	1	512	16384	1	1	16384
Decoder_4	26	1	1	1280	40960	1	1	40960
Decoder_1	27	1	1	64	2048	1	1	2048
Conv2	28	3	3	304	87552	3	3	87552
Conv3	29	3	3	256	73728	3	3	73728
prediction	30	3	3	256	73728	3	3	73728

Table A.1.: Weight Buffer Requirement in FPGA local memory

A.1.2. Latency Performance and DRAM access after Tiling the accelerator with configuration $16 \times 32 \times 4$

Block	Layer	Static Tiling				2D Tiling			
		T_{ox}	T_{oy}	DRAM access[MB]	Latency [ms]	T_{ox}	T_{oy}	DRAM access[MB]	Latency [ms]
conv_1_1	1	1	1	403.155	131.212	80	1	134.61	50.022
Res2_1	2	1	1	223.1464	55.216	10	10	24.308	11.037
Res2_1	3	1	1	223.14656	101.503	10	10	24.308	11.067
Res2_2	4	1	1	223.143936	55.219	10	10	24.308	11.045
Res2_2	5	1	1	223.14656	101.591	10	10	24.308	11.053
Res3_1	6	1	1	33.041	27.677	10	10	28.017	6.807
Res3_1	7	1	1	91.53	27.66	10	10	36.498	8.751
Res3_1	8	1	1	217.756	53.693	10	10	23.848	10.273
Res3_2	9	1	1	217.756	38.185	10	10	23.848	10.256
Res3_2	10	1	1	217.756	53.69	10	10	23.848	10.293
Res4_1	11	1	1	31.036	13.924	10	10	28.018	6.523
Res4_1	12	1	1	87.2544	19.308	10	10	35.846	8.479
Res4_1	13	1	1	214.357	37.986	10	10	22.946	10.11
Res4_2	14	1	1	214.357	38.055	10	10	22.946	10.113
Res4_2	15	1	1	214.357	38.033	10	10	22.946	10.103
Res5_1	16	1	1	143.529	27.681	10	10	34.413	8.402
Res5_1	17	1	1	668.991	75.699	1	1	668.991	75.694
Res5_1	18	1	1	1337.9829	150.51	1	1	1337.98	150.515
Res5_2	19	1	1	1337.9829	150.504	1	1	1337.98	150.515
Res5_2	20	1	1	1337.9829	150.513	1	1	1337.98	150.499
ASPP1	21	1	1	143.529472	25.609	10	10	34.4144	8.208
Decoder4_1	22	1	1	71.7648	14.069	10	10	17.206	4.433
ASPP2	23	1	1	594.542	75.928	1	1	594.542	75.897
Decoder4_2	24	1	1	72.647	25.597	10	10	17.2067	4.541
ASPP3	25	1	1	490.7335	75.651	1	1	490.733	75.665
Decoder4_3	26	1	1	72.646	25.614	10	10	17.2067	4.586
ASPP4	27	1	1	396.361	75.468	1	1	396.361	75.473
Decoder4_4	28	1	1	72.645	25.559	10	10	17.2067	4.409
ASPP5_Average	29	1	1	7.237	22.582	1	1	7.2376	22.571
ASPP5_Conv	30	1	1	0.010048	0.627	1	1	0.010048	0.575
ASPP5_Upsample	31	1	1	0.000512	13.915	1	1	0.000512	13.873
Decoder4_5	32	1	1	72.645	25.558	10	10	17.207	4.426
Upsample_Decoder4	33	1	1	1.8432	227.08	1	1	1.8432	227.099
Conv2_1	34	1	1	3511.47308	602.661	10	10	388.932	155.715
Decoder1	35	1	1	76.915	55.409	10	10	17.5719	5.604
Conv2_2	36	1	1	673.077	414.066	10	10	72.9244	34.542
Conv3	37	1	1	3511.473	602.694	10	10	388.931	155.68
Prediction	38	1	1	438.934	75.627	10	10	48.616	19.754
Total				**17869.893**	**3731.573**			**7746.114**	**1614.608**

Table A.2.: 2D Tiling benefits in Latency and DRAM access for restructured DeepLabV3+ network on FPGA [$16 \times 32 \times 4$]

A.2. Linear Regression Graphs of Hardware Model Latency for Special Layers of DeepLab

A.2.1. AveragePool Layer

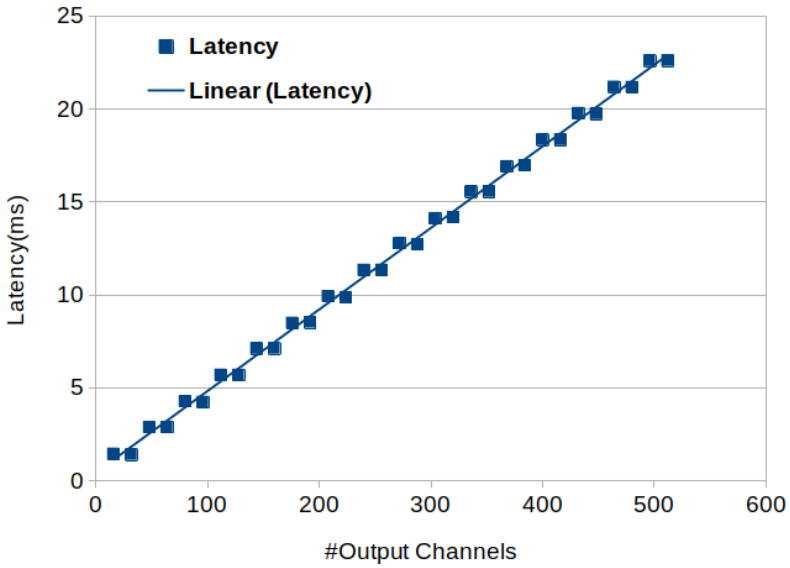

Figure A.1.: Linear Regression projection of hardware model based latency prediction for average pooling kernel

A.2.2. Upsampling Layer (Layer 31)

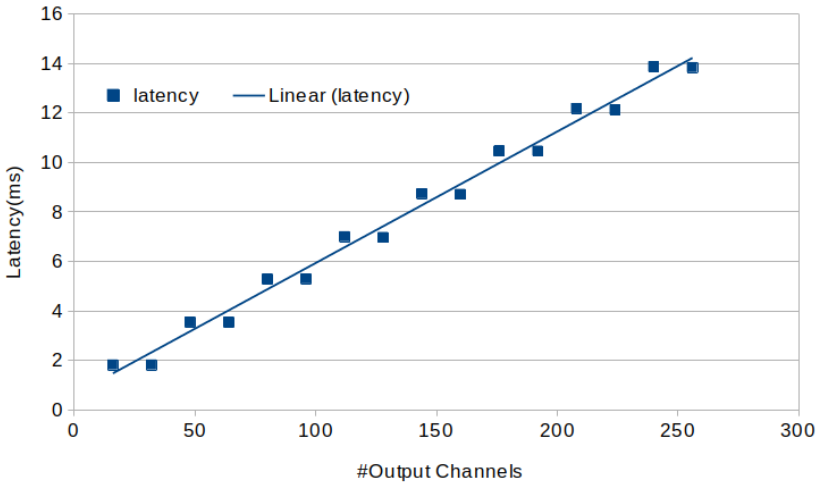

Figure A.2.: Linear Regression projection of hardware model based latency prediction for upsampling kernel

A.3. General Terminologies of Genetic Algorithm

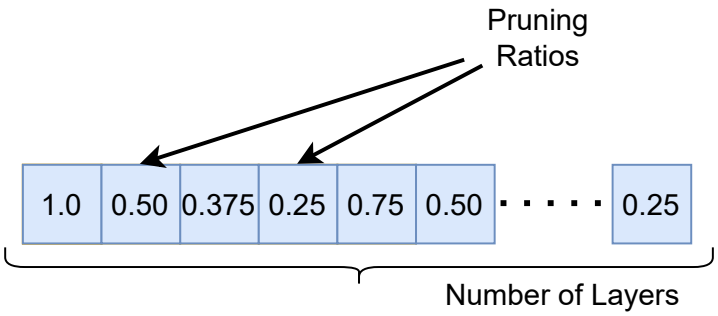

Figure A.3.: Encoding individuals

- **Crossover:** Cross-over is the process where, according to the crossover probability p_c, consecutive individuals in the population are mated to produce children. The mating can be performed through various techniques. *One-point crossover* and *two-point crossover*, illustrated in figure A.4, are the simplest

and the most common ones. The cross-over sites are selected at random and the genes are exchanged between parent individuals to produce offsprings or new individuals.

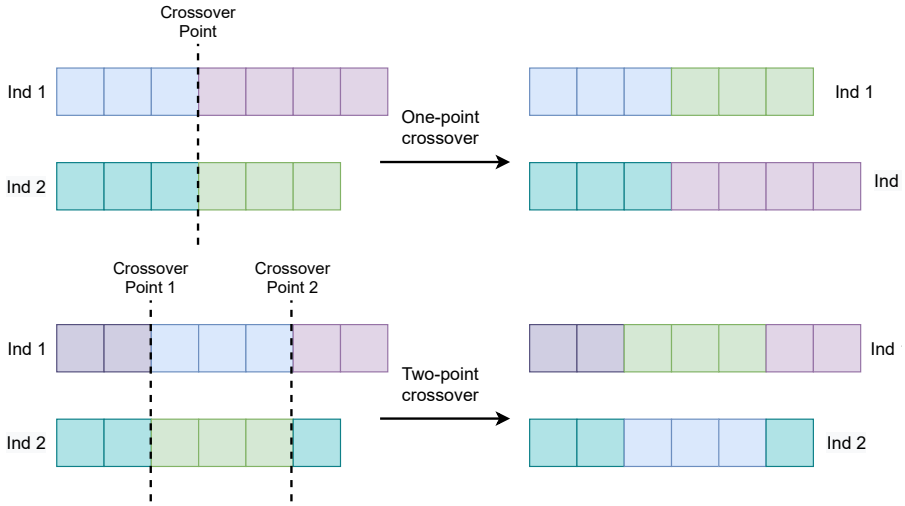

Figure A.4.: One-point and Two-point Crossover mechanisms

- **Mutation:** Mutation is the process of inserting random genes in offspring to maintain the diversity in population. The individuals produced after cross-over undergo mutation, with a probability p_m. The figure A.5 illustrates Replace mutation and Swap mutation. While replace mutation replaces one value of genotype in the individual, swap mutation swaps two values of the genotype with each other.

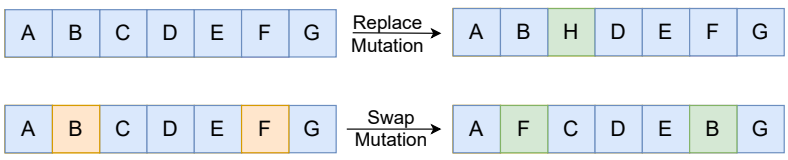

Figure A.5.: Replace and Swap Mutation mechanisms

A.4. DSP efficiency and Throughput of different configurations

Layer	#GOPs	$(16 \times 16 \times 1)$		$(16 \times 32 \times 1)$		$(16 \times 32 \times 4)$	
		Eff.	throughput	Eff.	throughput	Eff.	throughput
conv1_1	4.34	18.8 %	13.232	18.8 %	24.113	16.4 %	33.022
Res2_1:1	4.246	100 %	68.170	100 %	98.282	75 %	76.9
Res2_1:2	4.246	100 %	67.990	100 %	53.565	75 %	41.842
Res2_2:1	4.246	100 %	68.181	100 %	98.226	75 %	76.939
Res2_2:2	4.246	100 %	67.971	100 %	53.578	75 %	41.844
Res3_1:short	0.235	100 %	49.194	100 %	10.870	25 %	8.532
Res3_1:1	2.123	100 %	67.753	100 %	97.923	75 %	76.777
Res3_1:2	4.246	100 %	81.926	100 %	93.469	75 %	79.130
Res3_2:1	4.246	100 %	82.537	100 %	150.269	75 %	111.188
Res3_2:2	4.246	100 %	81.811	100 %	93.428	75 %	79.1
Res4_1:short	0.235	100 %	25.159	100 %	21.324	25 %	16.864
Res4_1:1	2.123	100 %	81.967	100 %	148.871	75 %	109.850
Res4_1:2	4.246	100 %	92.973	100 %	168.323	75 %	111.730
Res4_2:1	4.246	100 %	92.984	100 %	168.207	75 %	111.712
Res4_2:2	4.246	100 %	92.945	100 %	168.515	75 %	111.749
Res5_1:short	0.943	100 %	45.580	100 %	43.325	25 %	34.172
Res5_1:1	8.493	100 %	93.292	100 %	169.214	75 %	112.178
Res5_1:2	16.986	100 %	99.414	100 %	180.529	75 %	112.859
Res5_2:1	16.986	100 %	99.444	100 %	180.596	75 %	112.871
Res5_2:2	16.986	100 %	99.441	100 %	180.556	75 %	112.861
ASPP1	0.943	100 %	42.358	100 %	79.768	25 %	36.884
Decoder4_1	0.471	100 %	45.037	100 %	42.736	25 %	33.656
ASPP2	8.493	100 %	98.503	100 %	180.210	75 %	111.828
Decoder4_2	0.471	100 %	43.438	100 %	20.390	25 %	18.459
ASPP3	8.493	100 %	92.343	100 %	172.903	75 %	112.256
Decoder4_3	0.471	100 %	43.793	100 %	20.471	25 %	18.432
ASPP4	8.493	100 %	96.612	100 %	178.821	75 %	112.580
Decoder4_4	0.471	100 %	43.456	100 %	20.479	25 %	18.465
ASPP5_Average	0.025	100 %	1.607	100 %	1.462	100 %	1.143
ASPP5_Conv	0.0003	100 %	0.538	100 %	0.587	25 %	0.472
ASPP5_Upsample	0.028	100 %	5.947	100 %	2.619	100 %	2.055
Decoder4_5	0.471	100 %	43.636	100 %	20.464	25 %	18.479
Upsample	0.468	100 %	5.850	100 %	2.637	100 %	2.062
Conv2_1	67.947	100 %	93.167	100 %	170.137	75 %	112.743
Decoder1	0.353	100 %	51.620	**75 %**	8.172	18.75 %	6.383
Conv2_2	12.740	100 %	61.004	100 %	38.543	75 %	30.779
Conv3	67.947	100 %	93.169	100 %	170.091	75 %	112.75
Prediction	5.042	59.375 %	55.276	59.375 %	100.612	44.531 %	66.709
all	**295.258**	**92.996** %	**78.231**	**92.962 %**	**108.120**	**68.229** %	**79.135**

Table A.3.: DSP efficiency and Throughput in GOPS for different configurations $[(P_{if} \times P_{of} \times P_{kx} = 16 \times 16 \times 1)$, $(P_{if} \times P_{of} \times P_{kx} = 16 \times 32 \times 1)$ and $(P_{if} \times P_{of} \times P_{kx} = 16 \times 32 \times 4)]$ of segmentation based Accelerator for each layer of the DeepLabV3+ with no tiling.

Bibliography

[1] Zhong-Qiu Zhao, Peng Zheng, Shou tao Xu, and Xindong Wu. Object detection with deep learning: A review, 2019.

[2] H. Zhang, D. Liu, and Z. Xiong. Convolutional neural network-based video super-resolution for action recognition. In *FG*, 2018.

[3] D. Guiming, W. Xia, W. Guangyan, Z. Yan, and L. Dan. Speech recognition based on convolutional neural networks. In *2016 IEEE International Conference on Signal and Image Processing (ICSIP)*, 2016.

[4] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 2018.

[5] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[7] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *ISSCC*, 2016.

[8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGPLAN Not.*, 2014.

[9] NVIDIA. Nvidia deep learning accelerator. 2018.

[10] Linnan Wang, Wei Wu, Jianxiong Xiao, and Yang Yi. Large scale artificial neural network training using multi-gpus, 2015.

[11] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason 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.

[12] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[13] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.

[14] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *ECCV*, 2018.

[15] Arria 10 FPGA Development Kit Overview.

[16] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks, 2020.

[17] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference, 2019.

[18] Etienne Dupuis, David Novo, Ian O'Connor, and Alberto Bosio. Fast exploration of weight sharing opportunities for cnn compression, 2021.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[20] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and H.P. Graf. Pruning filters for efficient convnets. 08 2016.

[21] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks, 2017.

[22] Mengqi Yu, Hongzhi Huang, Hong Liu, Shuyi He, Fei Qiao, Li Luo, Fugui Xie, Xin-Jun Liu, and Huazhong Yang. Optimizing fpga-based convolutional encoder-decoder architecture for semantic segmentation. In *2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, pages 1436–1440, 2019.

[23] Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-cheung Ng, Yang Chu, and Wayne LUK. Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), December 2018.

[24] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[25] Khronos OpenCL Working Group. The opencl specification.

[26] Haoxing Ren. A brief introduction on contemporary high-level synthesis. In *2014 IEEE International Conference on IC Design Technology*, pages 1–4, 2014.

[27] Shuanglong Liu and Wayne Luk. Towards an efficient accelerator for dnn-based remote sensing image segmentation on fpgas. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 187–193, 2019.

[28] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[29] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *ArXiv*, abs/1505.00853, 2015.

[30] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2015.

[31] Image classification on imagenet, https://paperswithcode.com/sota/image-classification-on-imagenet.

[32] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807, 2017.

[33] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[34] K. He, X. Zhang, S. Ren, et al. Deep residual learning for image recognition. In *CVPR*, 2016.

[35] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[36] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J. Crowley, Björn Franke, Amos Storkey, and Michael O'Boyle. Performance aware convolutional neural network channel pruning for embedded gpus. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–34, 2019.

[37] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[38] E. Nurvitadhi et al. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *FPT*, Dec. 2016.

[39] U. Hatnik and S. Altmann. Using modelsim, matlab/simulink and ns for simulation of distributed systems. In *Parallel Computing in Electrical Engineering, 2004. International Conference on*, pages 114–119, 2004.

[40] Ahmed Mzid. Design and implementation of an opencl-based efficient accelerator for convolutional neural networks. *Research practice*, 2020.

[41] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[42] T. Tian, X. Jin, L. Zhao, X. Wang, J. Wang, and W. Wu. Exploration of memory access optimization for fpga-based 3d cnn accelerator. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020.

[43] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *CoRR*, abs/1606.00915, 2016.

[44] Liang-Chieh Chen, G. Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *ArXiv*, abs/1706.05587, 2017.

[45] Aäron van den Oord, S. Dieleman, H. Zen, K. Simonyan, Oriol Vinyals, A. Graves, Nal Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. In *SSW*, 2016.

[46] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, 2017.

[47] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, F. Corradi, A. Linares-Barranco, Shih-Chii Liu, and T. Delbruck. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Transactions on Neural Networks and Learning Systems*, 30:644–656, 2019.

[48] Dong Wang, Ke Xu, and Diankun Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017.

[49] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An opencl™ deep learning accelerator on arria 10. In *FPGA '17*, 2017.

[50] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018.

[51] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 696–701, 2014.

[52] Jouppi Et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[53] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.

[54] Lukas Cavigelli and Luca Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, 2017.

[55] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[56] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[57] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015.

[58] Liang-Chieh Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. *CoRR*, abs/1412.7062, 2015.

[59] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[60] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html, 2012.

[61] Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-Cheung Ng, Yang Chu, and W. Luk. Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11:1 – 22, 2018.

[62] Mengqi Yu, Hongzhi Huang, Hong Liu, Shuyi He, Fei Qiao, Li Luo, Fugui Xie, Xin-Jun Liu, and Huazhong Yang. Optimizing fpga-based convolutional encoder-decoder architecture for semantic segmentation. In *2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, pages 1436–1440, 2019.

[63] Gabriel J. Brostow, Jamie Shotton, Julien Fauqueur, and Roberto Cipolla. Segmentation and recognition using structure from motion point clouds. In *ECCV (1)*, pages 44–57, 2008.

[64] Hongzhi Huang, Yakun Wu, Mengqi Yu, Xuesong Shi, F. Qiao, Li Luo, Qi Wei, and Xinjun Liu. Edssa: An encoder-decoder semantic segmentation networks accelerator on opencl-based fpga platform. *Sensors (Basel, Switzerland)*, 20, 2020.

[65] Song Han, Huizi Mao, and W. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *arXiv: Computer Vision and Pattern Recognition*, 2016.

[66] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision (ECCV)*, 2018.

[67] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019.

[68] Tianzhe Wang, K. Wang, Han Cai, Ji Lin, Zhijian Liu, and Song Han. Apq: Joint search for network architecture, pruning and quantization policy. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2075–2084, 2020.

[69] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *ArXiv*, abs/1812.00332, 2019.

[70] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, V. Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. *ArXiv*, abs/1804.03230, 2018.

[71] Altera(Intel). Implementing fpga design with the opencl standard.

[72] Altera(Intel). Intel® arria® 10 device overview.

[73] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[74] Liang-Chieh Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40:834–848, 2018.

[75] Dong Wang, Ke Xu, and Diankun Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282, 2017.

[76] Pierpaolo Mori. Winograd aware quantized neural network acceleratordesign. *Master Thesis*, 2020.

[77] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

Confirmation

Herewith I, Saptarshi Mitra, confirm that I independently prepared this work. No further references or auxiliary means except those declared in this document have been used.

Munich, July 1, 2021

.....
Saptarshi Mitra