ACCELERATING AWARE MACHINE LEARNING ALGORITHMS DESIGN AND VERFICATION

By

Ahmed Tarek Mostafa

Abdallah Mohamed Said

Amr Adel Mohamady

Amr Mohamed Eid

Fatma Khaled Mohamed

Farida Khaled Mohamed

A graduation project sponsored by Mentor Graphics ICVS

Under the Supervision of

Associate Prof. Hassan Mostafa And **Dr. Eman El Mandouh**

A Graduation Project thesis

Submitted to the Faculty of Engineering at Cairo University

in Partial Fulfillment of the Requirements for the Degree of Bachelor of

Science in Electronics and Communications Engineering

Faculty of Engineering, Cairo University Giza, Egypt

August 2020

TABLE OF CONTENTS

TABLE OF FIGURES

TABLES

ABSTRACT

The Convolutional Neural Network gains an increasing importance nowadays because it enables machines to interact with surrounding environment and paving the way to computer vision applications to detect, classify and take decisions based on this data in all aspects of life. it is noted that GPUs have high power consumption and relatively large area making them unable to fit in mobile devices. On the other hand, FPGA implementations of CNN architectures have higher speed compared to GPUs and CPUs due to the parallel nature of FPGAs which can be used in real time applications. The purpose of this project is to implement Squeeze-Net CNN architecture on FPGA and achieving a very high speed with acceptable accuracy to contribute in nowadays trends towards machine learning applications in Automotive, security and others fields.

CHAPTER 1. INTRODUCTION

1.1 MOTIVATION

Artificial Intelligence has been witnessing a monumental growth in bridging the gap between the capabilities of humans and machines. Researchers and enthusiasts alike, work on numerous aspects of the field to make amazing things happen. One of many such areas is the domain of Computer Vision. The agenda for this field is to enable machines to view the world as humans do, perceive it in a similar manner and even use the knowledge for a multitude of tasks such as Image & Video recognition, Image Analysis & Classification, Media Recreation, Recommendation Systems, Natural Language Processing, etc. The advancements in Computer Vision with Deep Learning have been constructed and perfected with time, primarily over one particular algorithm — a **Convolutional Neural Network**.

A **Convolutional Neural Network (Conv-Net/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a Conv-Net is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, Conv-Nets have the ability to learn these filters/characteristics.

The major difference between a traditional Artificial Neural Network (ANN) and CNN is that only the last layer of a CNN is fully connected whereas in ANN, each neuron is connected to every other neuron as shown in figure.1. ANNs are not suitable for images because these networks lead to over-fitting easily due to the size of the images. Consider an image of size [32x32x3]. If this image is to be passed into an ANN, it must be flattened into a vector of 32x32x3 = 3072 rows. Thus, the ANN must have 3072 weights in its first layer to receive this input vector. For larger images, say [300x300x3], it results in a complex vector (270,000 weights), which requires a more powerful processor to process.

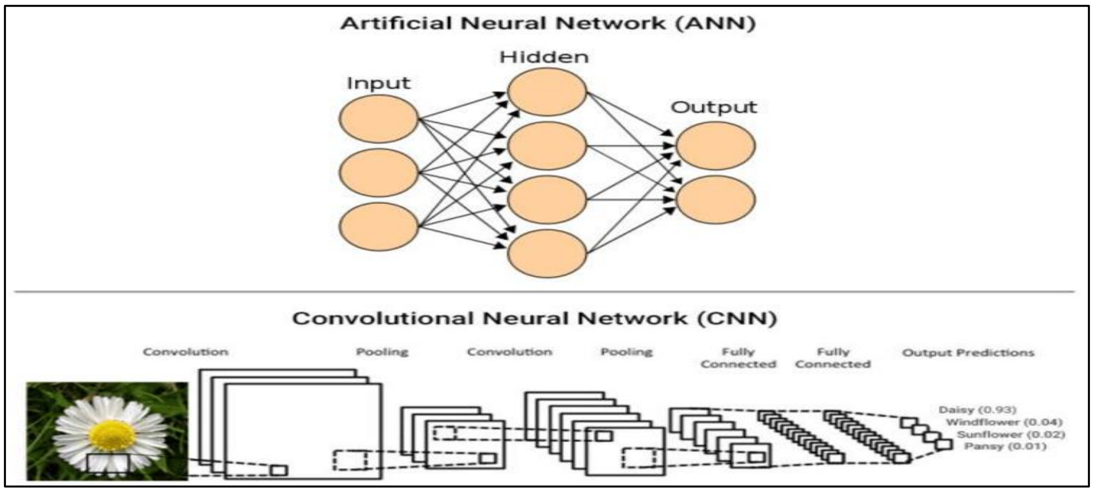


FIGURE 1: ANNS VS CNNS

CNNs consist of a stack of layers that takes in an input image, perform a mathematical operation (non-linear activation function such as RELU, tanh) and predicts the class or label probabilities at the output. Instead of using standard handcrafted feature extraction methods, CNNs takes in the raw pixel intensity of the input image as a flattened vector. For example, a [30x30] color image will be passed as a 3- dimensional matrix to the input layer of CNN. CNN automatically learns complex features present in the image using the different layers which has "learnable" filters and combines the results of these filters to predict the class or label probabilities of the input image. Unlike an ANN, the neurons in a CNN layer are not connected to all other neurons, but connected only to a small region of neurons in the previous layer. The first layer might detect the lowest level features such as corners and edges in the image. The next subsequent layers might detect middle level features such as shapes and textures, and finally higher-level features such as structure of the plant or flower will be detected by higher layers in the network. This unique technique of building up from lower level features to higher level features in an image is what makes CNNs most useful in many applications. The next section (section 1.2) dives into convolutional neural networks and explains each layer in detail.

1.2-LAYERS OF CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks are constructed by stacking a number of generic network layers, which transform the input feature maps of dimension $(in * w_{in} * ch_{in})$. into output feature maps of dimension $(h_{out} * w_{out} * ch_{out})$. A typical CNN path consists of two parts:

- The feature extractor which extracts features across the CNN layers which are; Convolutional (Conv), Pooling (Pool) Rectified Linear Unit (RELU).

- The Classifier, which is implemented using fully connected layers, takes these features and decides on the output class.

In order to understand the proposed hardware implementation, the CNN detailed layers will be discussed in this section.

1.2.1. Convolution layer

The Conv layer is the main block of a CNN that does most of the computations. From [1] it works by dividing the image into small regions (known as receptive field) and convolving them with a specific filter (multiplying weights of the filter or kernel (weights) with corresponding receptive field elements), then sliding these filters over the input feature maps as shown in figure 2. Each of these weight filters can be thought of as feature identifiers.

The computation is given in (1), where M is the number of output feature maps (number of filters) of size E × E, C is the number of channels in Input feature maps, and R × R is the size of the Filter.

$$OUT[m][ho][wo] = b_i + \sum_{i=1}^{c}\sum_{k_h=0}^{R}\sum_{k_w=0}^{R} IN[i][h_o + k_h][w_o + k_w] * kernel[m][i][k_h][k_w] \quad (1)$$

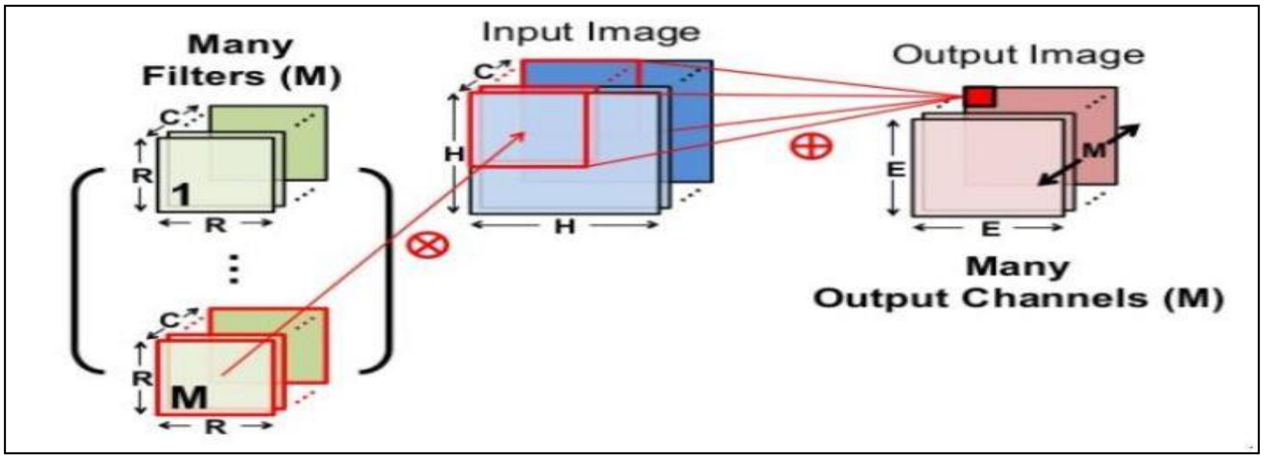


FIGURE 2: CONVOLUTION OPERATION

There are 2 parameters in conv layers which are listed below:

1. Filters:

The Conv layer's parameters consist of a set of learnable filters which work as feature detector (edges, simple colors, and curves).

2. Stride:

Stride is the number of pixels by which the filter matrix slides over the input matrix.

1.2.2. POOLING LAYER

The Pool layers (also called sub-sampling) do dimensionality reduction on each feature map, but keep the most important information. The depth of output feature maps is identical to that of input feature maps, while the size of each feature map scales down according to the size of the sub-sampling window (called also kernel) as shown in figure 3. By subsampling with some simple function; for example, average or maximum. Max-pooling being the most popular because the rate of change in the maximum value is very small compared to the rate of change of the average value of any receptive field (the window of the input feature map at which pooling is applied). Authors of [1] said that the use of pooling operation helps to extract a combination of features, which are invariant to translational shifts and small distortions. Also, pooling may reduce overfitting to training data.

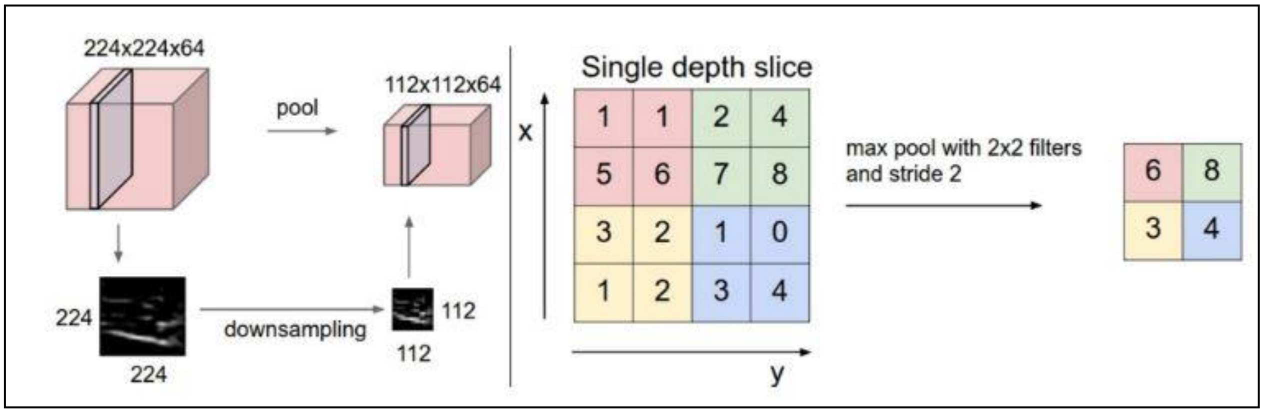


FIGURE 1: POOLING LAYER

1.2.3. ACTIVATION LAYER

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. It's important that activation functions be computationally efficient because they are calculated several times (thousands or even millions) for each data sample.

1.2.3.1. Binary step function

The binary step function outputs one if the input is positive and zero otherwise. Step function cannot support classifying the inputs into one of several categories. As they produce don't support multi-value output.

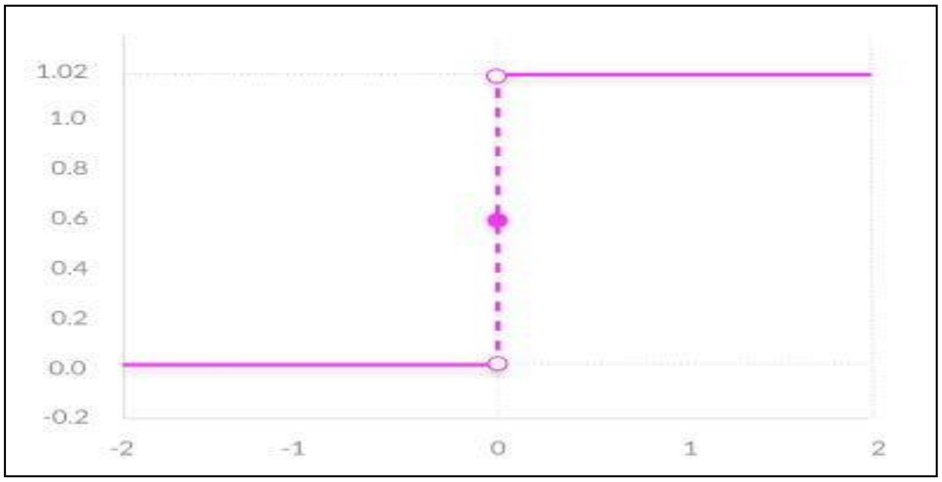


FIGURE 2: BINARY STEP ACTIVATION FUNCTION

1.2.3.2. Linear Activation functions

It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input as shown in figure 5. In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no.

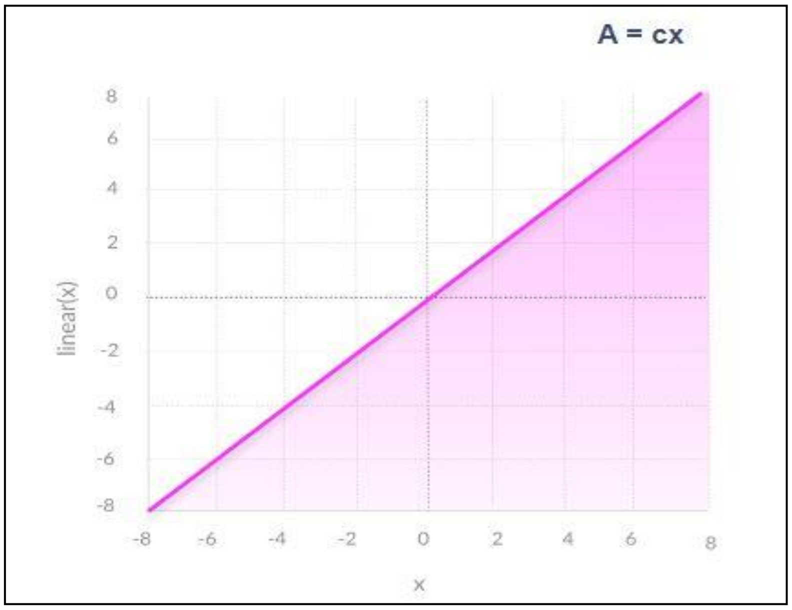


FIGURE 3: LINEAR ACTIVATION FUNCTION

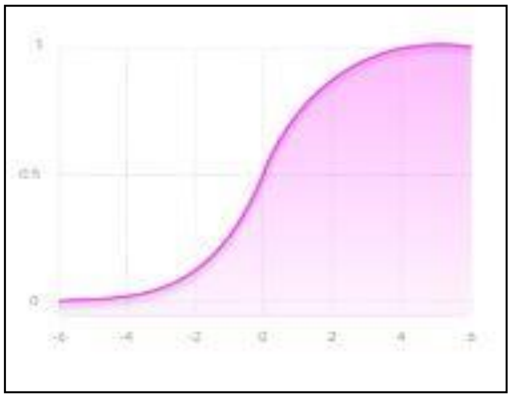
However, a linear activation function has two major problems:

1. Not possible to use backpropagation (gradient descent) to train the model since the derivative of the function is a constant, and has no relation to the input, X.

2. A linear activation function makes the neural network unable to fit non-linear data. They turn the neural network into just one layer; the last layer will be a linear function of the first layer (because a linear combination of linear functions is still a linear function).

1.2.3.3. NONLINEAR ACTIVATION FUNCTION

Networks use a non-linear activation functions which help them fit non-linear data such as images, video, audio, etc.



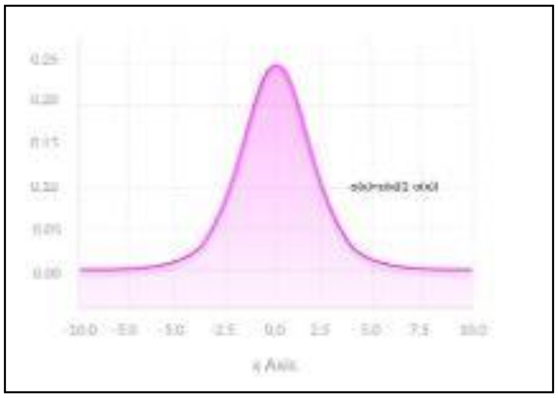


FIGURE 4: DERIVATIVE OF SIGMOID FUNCTION *FIGURE 5: SIGMOID FUNCTION*

Non-linear functions fix the problems of a linear activation function:

1. They have a derivative function which is related to the inputs and so allow backpropagation.
2. They allow stacking of multiple layers of neurons to create a deep neural network. Deep neural network with multiple hidden layers are needed to learn complex data with high levels of accuracy.

According to [2], there 7 types of Nonlinear Activation functions and here are the most popular nonlinear activation functions:

1. Sigmoid /Logistics functions

Advantages

- Smooth gradient, preventing "jumps" in output values.
- Output values between 0 and 1, normalizing the output of each neuron.
- Clear predictions—for X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1 or 0.

Disadvantages

- Vanishing gradient: For very high or very low input values, the derivative is almost zero, this can result in the network refusing to learn, or being too slow to reach an accurate prediction.
- Outputs not zero centered**.**
- Computationally expensive due to the exponential function.

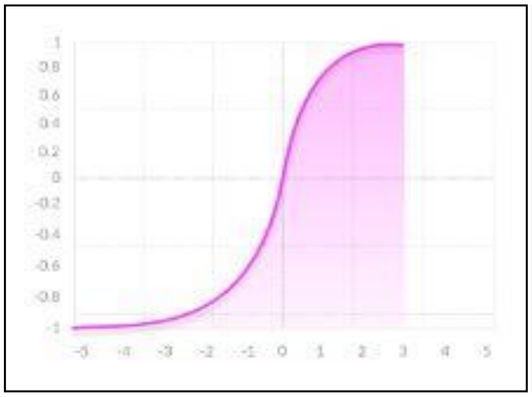
2. Tanh /Hyperbolic Tangent

Advantages

- Zero centered: which make it easier to model inputs that have negative, neutral, and positive values.
- Otherwise like the sigmoid function.

Disadvantages

- Like the Sigmoid function



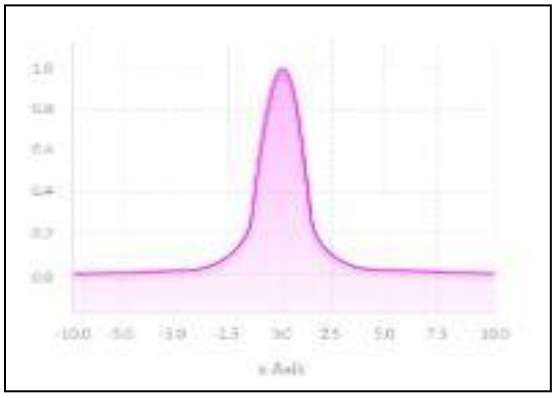


FIGURE 6: DERIVATIVE OF TANH FUNCTION

FIGURE 7: TANH FUNCTION

3. RELU (RECTIFIED LINEAR UNIT)

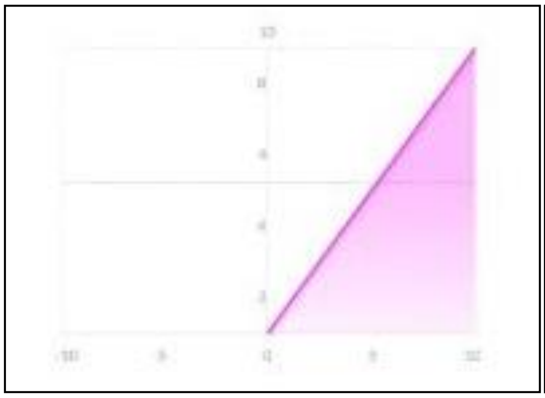
The RELU is a piece-wise linear function. It outputs the same input if it's positive and outputs zero for negative values.

Advantages

- Computationally efficient
- No vanishing gradient problem for positive inputs

Disadvantages

- For negative inputs the gradient becomes zero, the network cannot perform backpropagation and cannot learn. This issue is called the dying RELU.



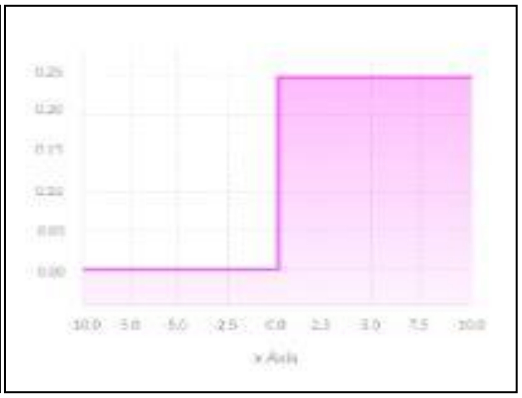


FIGURE 8: DERIVATIVE OF RELU FUNCTION *FIGURE 9: RELU FUNCTION*

RELU and its variants are preferred over others activations as it helps in overcoming the vanishing gradient problem.

1.2.4. Normalization Layer (Batch Normalization)

Neural networks learn slowly if the distribution of their input's changes over time. This issue is called the internal covariance shift. Batch normalization addresses this issue by bringing the values to zero mean and unit variance. It then multiplies the normalized results by a learnable parameter (new variance) and add a learnable parameter to it (new mean) and so gives the network the ability to choose the suitable distributions. It also smoothens the flow of gradient and acts as a regulating factor, which improves generalization of the network without relying on dropout. [1]

Batch normalization for transformed feature map T_l^k is shown in equation (2).

$$N_l^k = \frac{T_l^k}{\sigma^2 + \sum_i T_i^K} \qquad (2)$$

In equation (2), N_l^k represents normalized feature map, where T_l^k is input feature map and σ depicts variation in feature map.

1.2.5 Drop out Layer

Dropout works by randomly skipping some units or connections in the network thus, introduces regularization within network, which ultimately improves generalization. In NNs, multiple connections that learn a non-linear relation are sometimes co-adapted, which causes overfitting.

1.2.6. Fully Connected Layer

Fully connected layer is mostly used at the end of the network for classification purpose. It takes input from the previous layer and analyses output of all previous layers globally. It makes a non-linear combination of selected features, which are used for the classification of data. Unlike pooling and convolution, it is a global operation [1].

1.3-Different CNN Architectures

In recent years, the world witnessed the birth of numerous CNNs. These networks have gotten so deep that it has become extremely difficult to visualize the entire model. We stop keeping track of them and treat them as black box models. This section is a visualization of 4 ***common*** CNN architectures and then in next section (section 1.4), Squeeze-Net is explained in detail. These illustrations provide a more compact view of the entire model of each architecture.

1.3.1-AlexNet (2012)

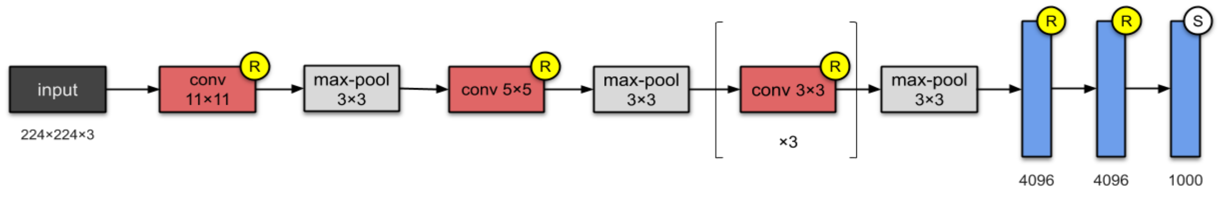


FIGURE 10: ALEXNET ARCHITECTURE

Alex-Net is one of most famous CNN architecture because it was the first CNN to win the ILSVRC in 2012.AlexNet consist of 5 conv layers and 3 max-pooling layers in addition to 3 fully-connected layers in the end of architecture as shown in figure 12. It has a huge size and 60 million parameters; Alex-Net was the first to implement:

1. Rectified Linear Units (RELUs) as activation functions.
2. Overlapping pooling in CNNs.

1.3.2-VGGNET (2014)

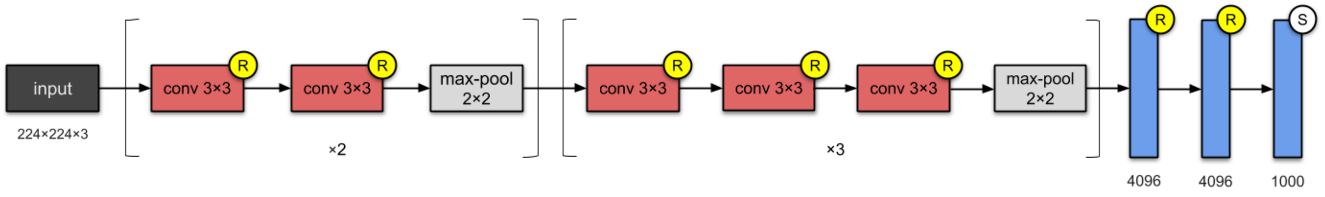


FIGURE 11: VGG-NET ARCHITECTURE

In this architecture the designers from Visual Geometry Group (VGG) depend on the most straightforward way to improve the accuracy of CNN is to go deeper and increase the number of layers and parameters.as shown in figure 14, VGG-Net has 13 convolutions layers and 5 max-pooling layers followed by 3 fully-connected layers in near to output. And it depends also on RELU the same activation function of Alex-Net. VGG-Net has **138M parameters** and takes up about 500MB of storage space.it has another version VGG-19 with a greater number of layers and higher accuracy.

1.3.3-GOOGLENET (INCEPTION V1)

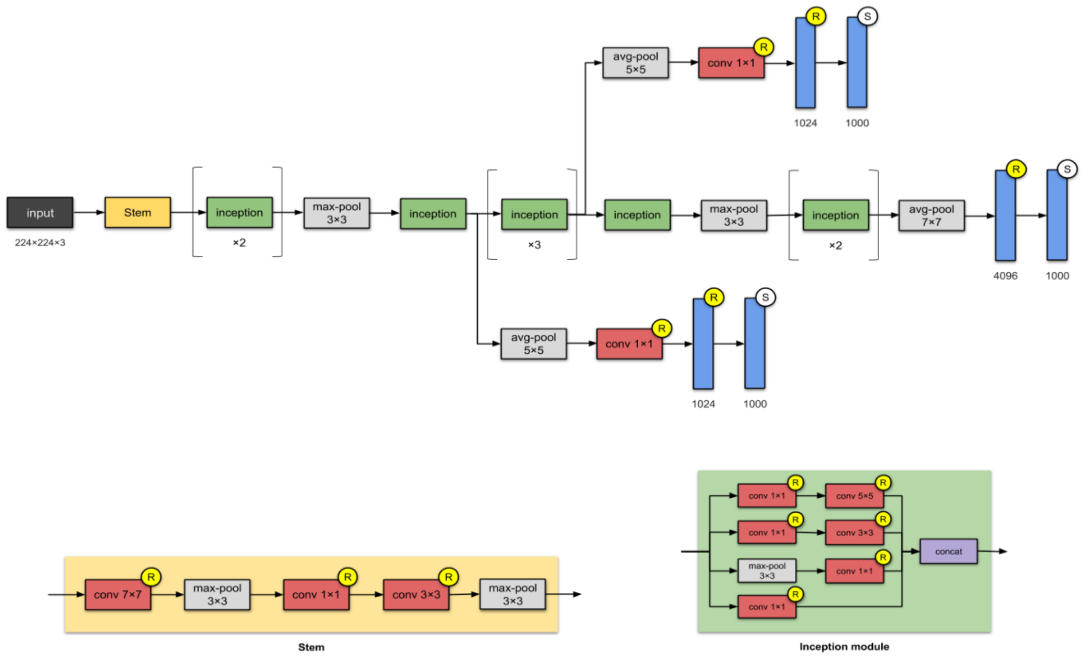


FIGURE 12: GOOGLE-NET ARCHITECTURE

This 22-layer architecture with **5M** parameters is called the Inception-v1, shown in figure 15. The main building block of this architecture is the “Inception modules”. Finding out how an optimal local sparse structure in a CNN can be approximated and covered by readily available dense components is the idea of the Inception module. It is needed to find the optimal local construction and to repeat it spatially. A layer-by layer construction in which one should analyze the correlation statistics of the last layer and grouping them in units with high correlation [6]. These groups form the units of the following layer and are linked to those in the previous one. The lower layer (the ones close to the input) consists of units which corresponds to a certain region in the input image and then grouped into filter banks. In these layers correlated units would concentrate in local regions. This means, this would end up with a lot of groups concentrated in a single region [6]. So, by using layers of 1x1 convolution there will be decreasing in number of patches over the network. In order to avoid patch alignment issues, The Inception architecture has been built only by filter sizes 1×1, 3×3 and 5×5.This means that the output of these layers is concatenated together, and form the input of the next stages. As pooling layers have been important for CNNs, the authors suggest that adding an alternative parallel pooling path in each stage will increase the accuracy (see figure 16(a)). As these “Inception modules” are stacked on top of each other, their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease suggesting that the ratio of 3×3 and 5×5 convolutions should increase as we move to higher layers. One big problem with the above modules, at least in this naive form,

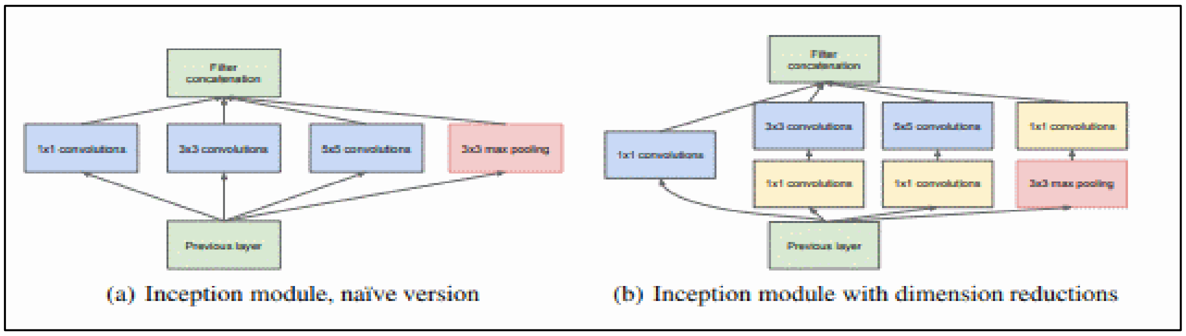


FIGURE 13: INCEPTION MODULE

is when we even use a reasonable number of 5x5 convolutions with large number of filters on the top of the network, this increases the number of the output channels from stage to another leading to a huge computational cost in few stages. To avoid these problems a second idea has been found from the proposed architecture by reducing the dimensions and applying dimensional projections when the computational cost increase. 1×1 convolution is used for reductions before the 3×3 and 5×5 convolutions. Moreover, 1x1 conv layers used for reduction, they also include rectified-linear functions, so they have been considered to have dual purpose. The final result is depicted in figure 15(b). In general, an Inception network is a network consisting of Inception modules stacked above each other, with stride 2 max-pooling layers to decrease the resolution to halve of its dimension. Traditional convolution has been used in lower layers while inception modules in higher ones for memory efficiency, this is not necessary. So, the main beneficial aspects of this architecture are: 1) Increasing the number of units at each stage significantly without an uncontrolled blow-up in computational complexity. 2) Dimension reduction allows for shielding the large number of input filters of the last stage to the next layer, first reducing their dimension before convolving over them with a large patch size. [8]

1.3.4 ResNet-50 (2015)

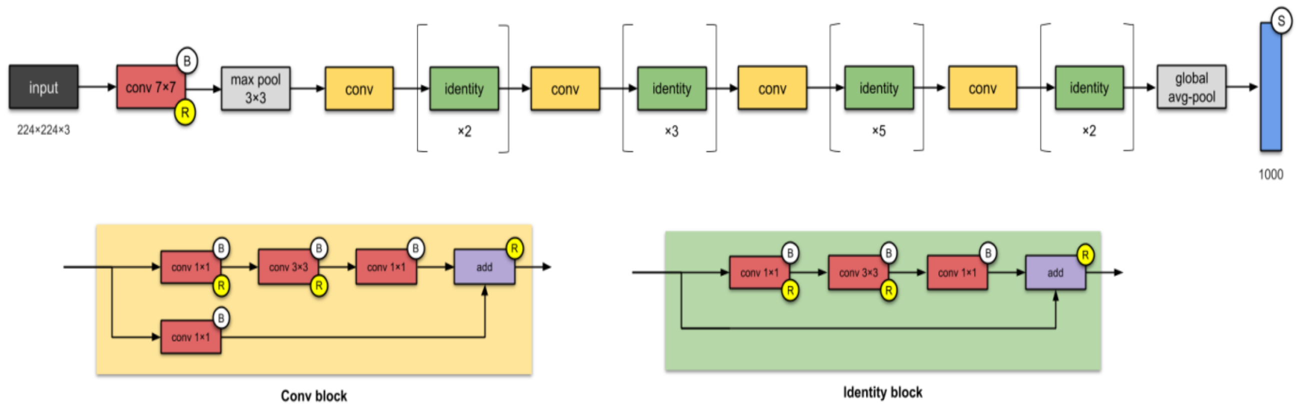


FIGURE 14: RES-NET ARCHITECTURE

Researches from Microsoft introduced another CNN architecture called ResNet-50 with 26 million parameters. As shown in figure 16, Res-Net has 2 basics blocks, the conv and identity blocks. It uses batch normalization and a new concept called skip connections to overcome on some problems happens when stacking layers. The most straightforward way to increase the performance is by going deeper and increase the number of layers. While Alex-Net had only 5 convolutional layers, the VGG network and Google-Net had 19 and 22 layers respectively. But in some point, accuracy saturates and might then degrade rapidly that happens because of that; The CNN gets more difficult to train because of notorious vanishing gradient problem. Deep networks are hard to train because of the as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small [9] as shown in Figure 17. So, finding another way to achieve better performance is necessary.

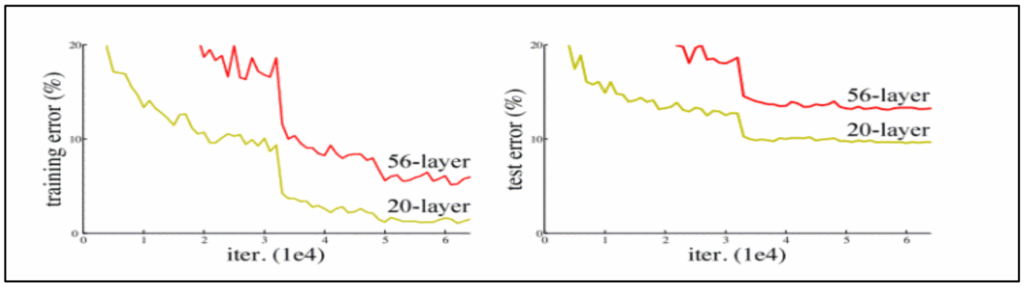


FIGURE 15: INCREASING NETWORK DEPTH LEADS TO WORSE PERFORMANCE

The main idea of Res-Net is introducing a called "identity shortcut connection" (sometimes called skip or residual connection) that feeds the next layer and the other that 2-3 layers away, as shown in Figure 18. This method helps to skip the layers that contributed in degradation and save the accuracy [9].

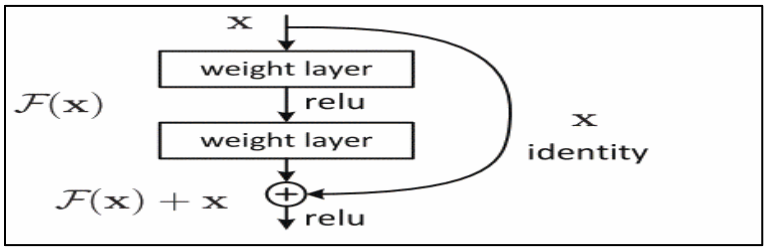

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

FIGURE 16: RESIDUAL BLOCK

- **Convent estimates for single element batch for memory consumption and FLOP counts for all CNNs [4]**

Table 1: Comparison between different architectures

CNN Arch.	Input size	Parameters memory	Features memory	Flops	Performance
Alex-Net	224x224	233 MB	3 MB	727 MFLOPs	41.80 / 19.20
VGG-Net	224x224	528 MB	58 MB	16 GFLOPs	28.50 / 9.90
Google-Net	224x224	51 MB	26 MB	2 GFLOPs	34.20 / 12.90
Res-Net	224x224	98 MB	103 MB	4 GFLOPs	24.60 / 7.70
Squeeze-Net	224x224	5 MB	30 MB	837 MFLOPs	41.90 / 19.58

1.4-SQUEEZE-NET

Recent researches on deep learning are focused primarily to achieve high level accuracy, smaller CNN architectures offer at least three advantages while maintaining the same accuracy or higher:

I. Require less communication during the training process.
II. Achieve low size model, Therefore, easily to export from cloud to a lot of applications
III. More feasible to deploy on FPGAs and other hardware with limited memory. To provide all of these advantages, we propose a small CNN architecture called Squeeze-Net.

Squeeze-Net achieves Alex-Net-level accuracy on ImageNet with 50x fewer parameters. Additionally, with model compression techniques, we are able to compress Squeeze-Net to less than 0.5MB (510× smaller than Alex-Net), it designed with three main strategies:

I. Replace 3x3 filters with 1x1 filters so the weight parameters are decreased nine times
II. Decreasing the number of input channels which is entered to 3*3 filters in expand layer to still maintain small number of parameters as it equal (number of input channels) * (number of filters) * (3*3) which is the objective of squeeze layer
III. Down sampling late in the network so that convolution layers have large activation maps, which can lead to higher classification accuracy

Strategies 1 and 2 are about decreasing the number of parameters in a CNN while attempting to preserve accuracy; Strategy 3 is about maximizing accuracy on a limited budget of parameters [12].

1.4.1 FIRE MODULES

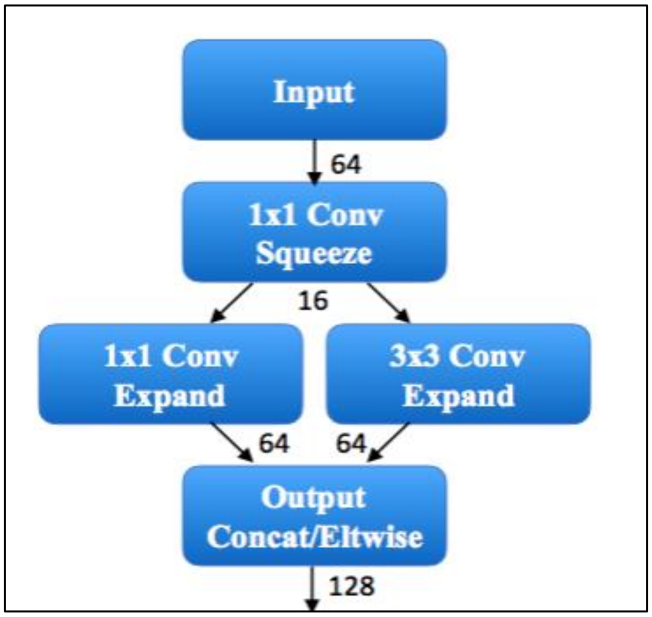


FIGURE 17: FIRE MODULE

A Fire module is the building block of Squeeze-Net, it contains squeeze convolution layer (which has only 1x1 filters), feeding into an expand layer which has a mix of 1x1 and 3x3 convolution filters, the objective of use of 1x1 filters in Fire modules is an application of Strategy 1 from Section 1.4. There are three tunable dimensions (hyperparameters) in a Fire module: s1x1, e1x1, and e3x3. In a Fire module, s1x1 is the number of filters in the squeeze layer (all 1x1), e1x1 is the number of 1x1 filters in the expand layer, and e3x3 is the number of 3x3 filters in the expand layer. When we use Fire modules, we set s1x1 to be less than (e1x1 + e3x3), so the squeeze layer helps to limit the number of input channels to the 3x3 filters, as per Strategy 2.

Bottleneck layers are used in fire module, the main idea is to reduce the size of the input tensor in a convolutional layer with kernels bigger than 1x1 by reducing the number of input channels. This technique helps in keeping the number of parameters, and thus the computational cost low, as shown in the example in figure 20.

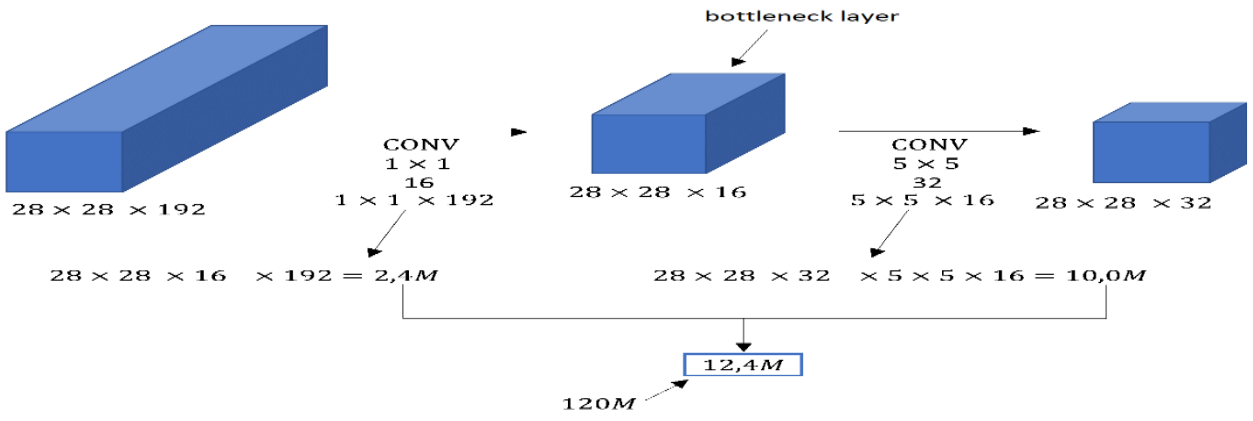


FIGURE 18: AN EXAMPLE SHOWS THE EFFECT OF THE BOTTLENECK LAYER ON THE COMPUTATION COST

1.4.2-Squeeze-Net Architecture

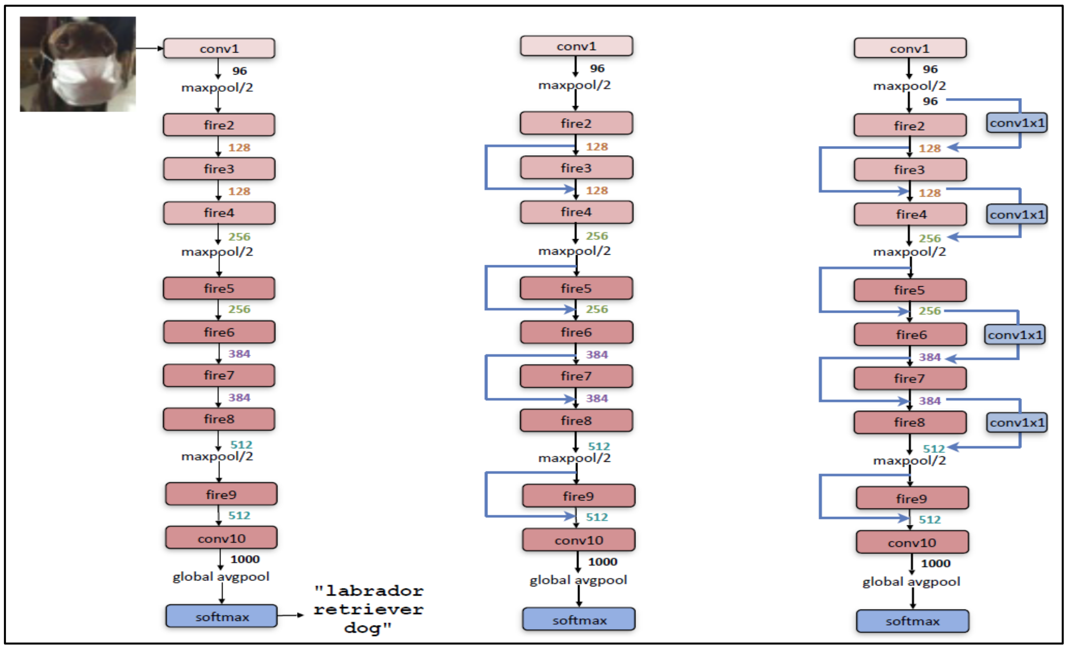
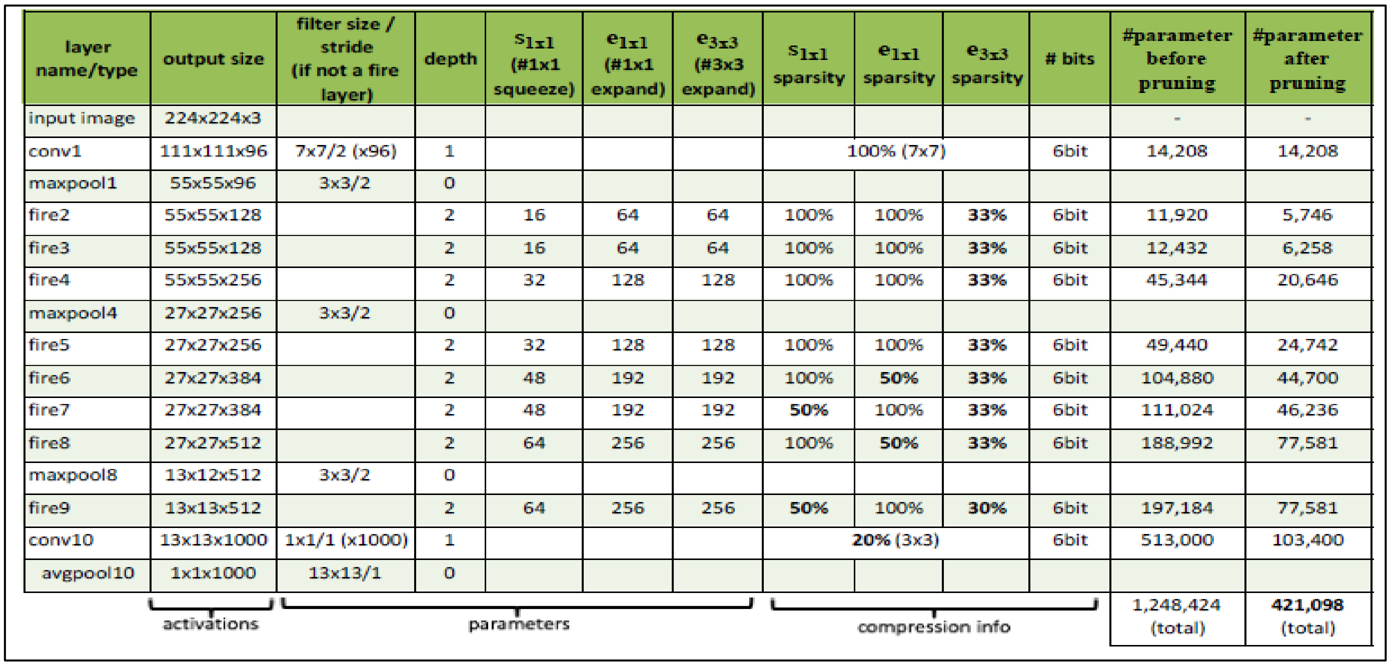


FIGURE 19: SQUEEZE-NET (LEFT), SQUEEZE-NET WITH SIMPLE BYPASS (MIDDLE), SQUEEZE-NET WITH COMPLEX BYPASS



layer name/type	output size	filter size / stride (if not a fire layer)	depth	s_{1x1} (#1x1 squeeze)	e_{1x1} (#1x1 expand)	e_{3x3} (#3x3 expand)	s_{1x1} sparsity	e_{1x1} sparsity	e_{3x3} sparsity	# bits	#parameter before pruning	#parameter after pruning
input image	224x224x3										-	-
conv1	111x111x96	7x7/2 (x96)	1					100% (7x7)		6bit	14,208	14,208
maxpool1	55x55x96	3x3/2	0									
fire2	55x55x128		2	16	64	64	100%	100%	**33%**	6bit	11,920	5,746
fire3	55x55x128		2	16	64	64	100%	100%	**33%**	6bit	12,432	6,258
fire4	55x55x256		2	32	128	128	100%	100%	**33%**	6bit	45,344	20,646
maxpool4	27x27x256	3x3/2	0									
fire5	27x27x256		2	32	128	128	100%	100%	**33%**	6bit	49,440	24,742
fire6	27x27x384		2	48	192	192	100%	**50%**	**33%**	6bit	104,880	44,700
fire7	27x27x384		2	48	192	192	**50%**	100%	**33%**	6bit	111,024	46,236
fire8	27x27x512		2	64	256	256	100%	**50%**	**33%**	6bit	188,992	77,581
maxpool8	13x12x512	3x3/2	0									
fire9	13x13x512		2	64	256	256	**50%**	100%	**30%**	6bit	197,184	77,581
conv10	13x13x1000	1x1/1 (x1000)	1					**20%** (3x3)		6bit	513,000	103,400
avgpool10	1x1x1000	13x13/1	0									
	activations			parameters				compression info			1,248,424 (total)	**421,098** (total)

FIGURE 20: SIMPLE SQUEEZE-NET ARCHITECTURE

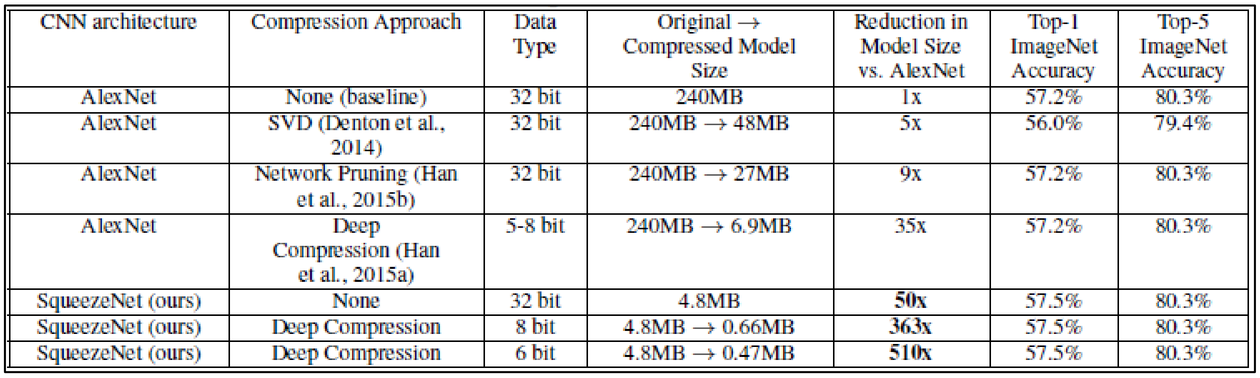
As shown in figure 22 & 23:

- **Simple Squeeze-Net**: begins with a standalone convolution layer (conv1), followed by 8 Fire modules (fire2–9), ending with a final conv layer (conv10).
- The number of filters per fire module is gradually increased from the beginning to the end of the network.
- Max-pooling with a stride of 2 is performed after layers conv1, fire4, fire8, and conv10.
- **Squeeze Net with simple bypass (Middle) and Squeeze Net with complex bypass (Right)**: The use of bypass is inspired by Res-Net.

1.4.3-Evaluation of Squeeze-Net

In figure 23, Squeeze-Net has been reviewed in the context of recent model compression results. There are ways able to compress the pertained model of Alex-Net to decrease the model size, SVD compresses it by factor of 5x while decreasing the accuracy of top-1 to 56% (Denton et al., 2014). Network pruning compresses it by factor of 9x while maintain the accuracy of ImageNet the same. Deep compression compresses it by factor of 35x while maintain the accuracy the same. Now squeeze-net achieves 50x reduction in model size while meeting top –1 accuracy and top –5 accuracy of Alex-net without any compression techniques. Using 33% sparsity 8-bit quantization this leads to 0.66MB model (363× smaller than 32-bit Alex-Net). Using 33% sparsity 6-bit quantization this leads 0.47MB model (510× smaller than 32-

bit Alex-Net) with the same accuracy of Alex-Net as shown in figure 24 [12] Compression (Han et al., 2015a) not only works well on CNN architectures with many parameters (e.g. Alex-Net and VGG), but it is also able to compress the already compact, fully convolutional Squeeze-Net architecture. Deep Compression compressed Squeeze-Net by 10× while preserving the baseline accuracy. In summary: by combining CNN architectural innovation (Squeeze-Net) with state-of-the-art compression techniques (Deep Compression), we achieved a 510× reduction in model size with no decrease in accuracy compared to the baseline. Finally, note that Deep Compression (Han et al., 2015b) uses a codebook as part of its scheme for quantizing CNN



CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	**50x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	**363x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	**510x**	57.5%	80.3%

FIGURE 21: COMPARING SQUEEZE-NET TO MODEL COMPRESSION APPROACHES

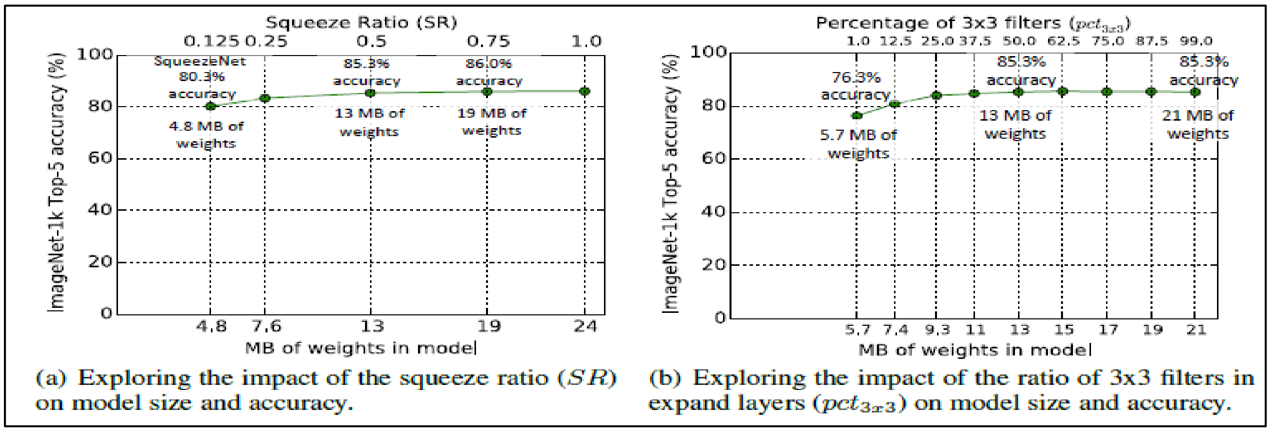


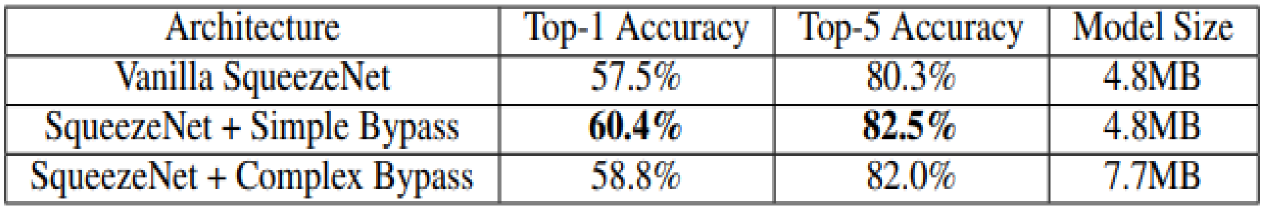
FIGURE 22: DIFFERENT HYPERPARAMETER VALUES FOR SQUEEZE-NET

1.4.4-HYPERPARAMETERS

In figure 25, at the left the Squeeze ratio (SR) is the ratio between the number of filters in squeeze layers and the number of filters in expands layers. By Increasing SR beyond 0.125 can further increase ImageNet top-5 accuracy from 80.3% (i.e. Alex-Net-level) with a 4.8MB model to 86.0% with a 19MB model. Accuracy plateaus at 86.0% with SR=0.75 (a 19MB model), and setting SR=1.0 further increases model size without improving accuracy, however, at the right the Percentage of 3×3 Filters (Right): Top-5 accuracy plateaus at 85.6% using 50% 3×3 filters, and further increasing the percentage of 3×3 filters leads to a larger model size but provides no improvement in accuracy on ImageNet. [12]

1.4.5-SQUEEZENET VARIANTS

As in figure 26, results of the accuracy of different architectures of squeeze-net like simple, simple bypass and complex bypass.



Architecture	Top-1 Accuracy	Top-5 Accuracy	Model Size
Vanilla SqueezeNet	57.5%	80.3%	4.8MB
SqueezeNet + Simple Bypass	**60.4%**	**82.5%**	4.8MB
SqueezeNet + Complex Bypass	58.8%	82.0%	7.7MB

FIGURE 23: SQUEEZE-NET ACCURACY AND MODEL SIZE USING DIFFERENT MICROARCHITECTURE CONFIGURATIONS

1.5-FIXED-POINT REPRESENTATION

Because of improvement in DCNN and increasing the number of layers and parameters to achieve better accuracy, methods to decrease computational complexity of floating-point arithmetic are highly needed. The fixed-point number system shows the best trade-off between accuracy and computational complexity in hardware-based applications. Fixed-point has two main advantages. firstly, the smaller hardware implementation of fixed point-based system allows for more modules to be instantiated in same area with same logic gates that increases the opportunities of parallelism and pipelining. Secondly, the smaller data representation of parameters or input pixels reduces the required memory, enabling larger CNN models to fit within the given memory capacity and reducing the power of memory-access because more data fit within same memory bandwidth. [13]

The representation of any number in fixed point consists of two parts IL and FL as shown in figure 28, the integer part. And fractional part respectively and another bit (S) is added to indicate the number sign.

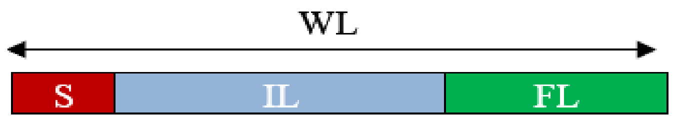


FIGURE 24: FIXED POINT NUMBER REPRESENTATION

The required bits of represent the result of addition of two numbers with same length WL is WL+1.and the required bits of represent the result of multiplication of two numbers with same length WL is 2*WL+1.The threats of using fixed-point number system such as manipulating overflow and saturation are extremely rare and occurs when all integer and fraction are saturated to either the lower or the upper limit of IL and FL. Results show a good performance of static fixed point for as low as 18-bit, however, when reducing the bit-width further, the accuracy starts to drop significantly. To overcome this problem dynamic fixed-point is used. Since the range of numbers is $[-2^{IL-1}, 2^{IL-1} - 2^{-FL}$ [and resolution is 2^{-FL} so increasing the FL will cover more numbers that helps to increase the accuracy of multiplication and addition and avoid vanishing of small parameters. Every layer has its suitable WL that may differ of next layer to reduce the loss. Ristretto is frameworks that choose the best dynamic fixed-point WL of trained model [14]. The WL can be 32-bits, 16-bits and 8-bits.the choose of suitable bit-width is tradeoff between memory footprint and accuracy loss. Performing deep learning inference using the squeeze net architecture on the ImageNet dataset, with 8-bit dynamic fixed-point weights and 8-bit dynamic fixed-point data, resulting in less than 1% degradation of accuracy [15].

Replacing 32-bits floating point with 8-bits fixed point for a hardware implementation, this reduces the size of multipliers by about one order of magnitude. Moreover, the required memory is reduced by 4–8X.it helps to hold 4–8X more parameters in on-chip buffers [15].

1.6-PARALLELISM

DCNN Applications in embedded systems are required to be fast and low power. Parallelism in CNN hardware accelerators offer opportunities to increase the throughput of operations and decrease the switching power of memory accessing. Here different methods of parallelism are discussed.

1.6.1 Inter Layer Parallelism

Data between the layers of CNN architecture are dependent so no layer can be executed before another and layers can't work in parallel. Squeeze-net has 10 fire modules; first fire module can perform its operations and delivers the output as input to second module. Now second fire module begins its operations without any dependency on pervious module so first fire module can process another input image without problems. This method called pipelining and can increase the total throughput of DCNN.

1.6.2 Inter Output Parallelism

Every output feature map is result of convolution between the input feature map and filter. All filters are independent and different so the output feature maps are totally independent of each other. The convolution between the input volume and different filters can be performed in parallel without problems as shown in figure 29. That can decrease the required cycles to access memory for different filters.

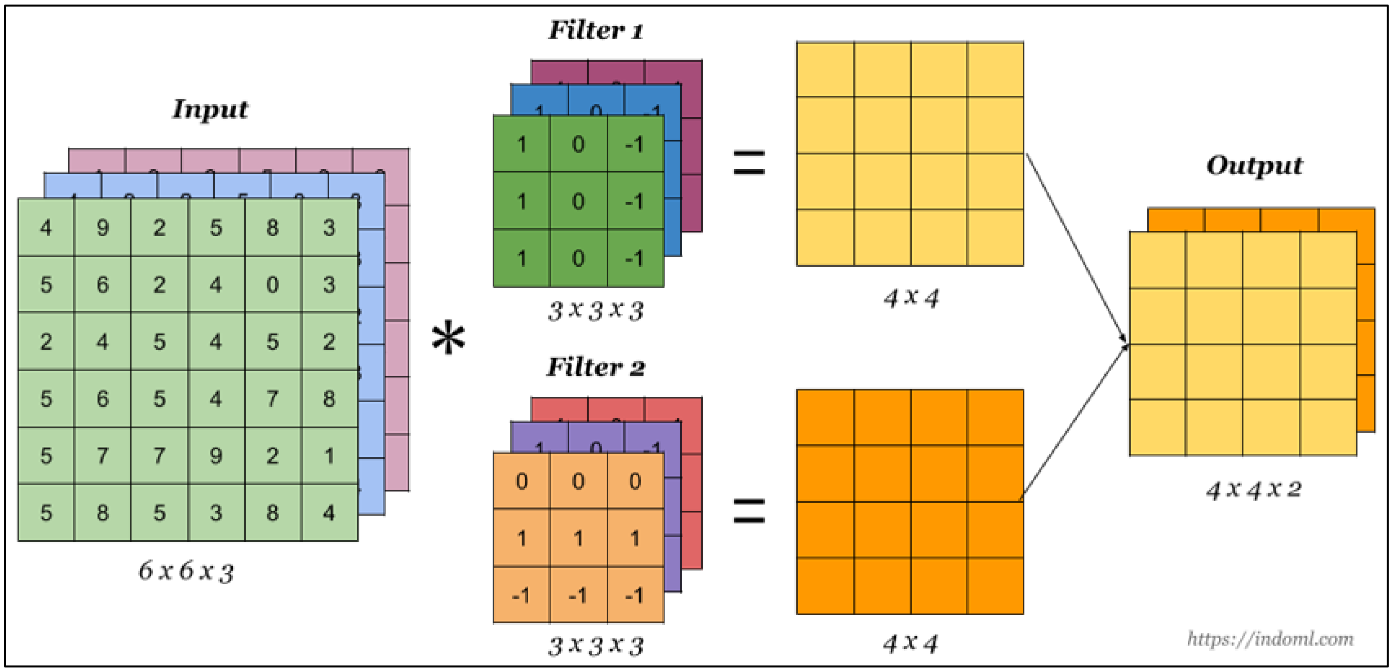


FIGURE 25: INTER OUTPUT PARALLELISM

1.6.3 Inter Kernel Parallelism

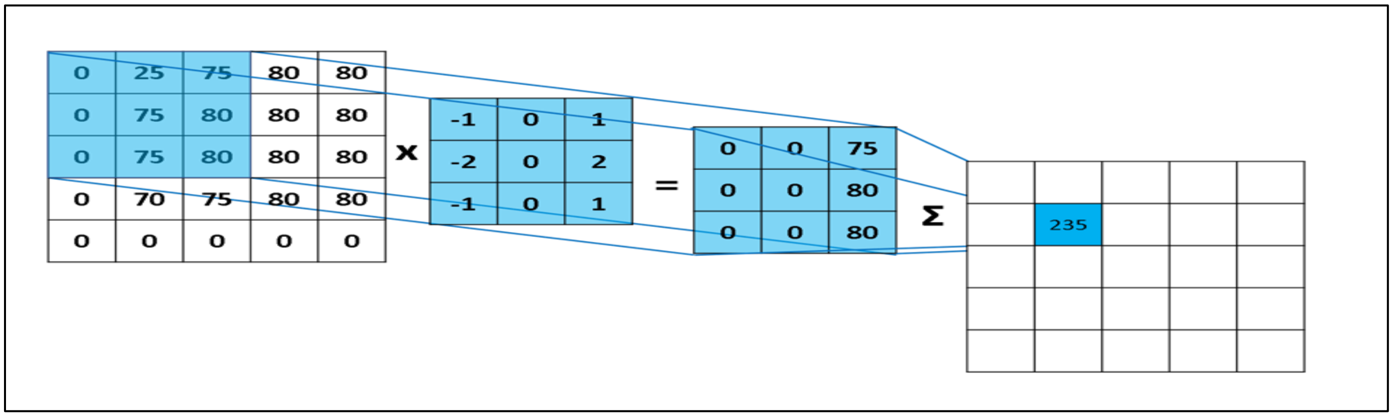


FIGURE 26: INTER KERNEL PARALLELISM

The output feature map is result of swapping filter on all input feature map and performing convolution. Each convolution is independent since input pixels are independent from each other so it is possible to compute all of output pixels in output feature map concurrently as shown in figure 28.

1.6.4 Intra Kernel Parallelism

The convolution is set of multiplication and addition operations. Every multiplication operation between the input pixel and corresponding parameter in filter and all of operations are independent and the addition operation of results from multiplication to calculate the output pixel can be performed in parallel also as shown in figure 29. Squeeze-net has 3x3 and 7x7 filters so parallelism the multiplication operations can increase the speed by x9 and x49 respectively in these layers.

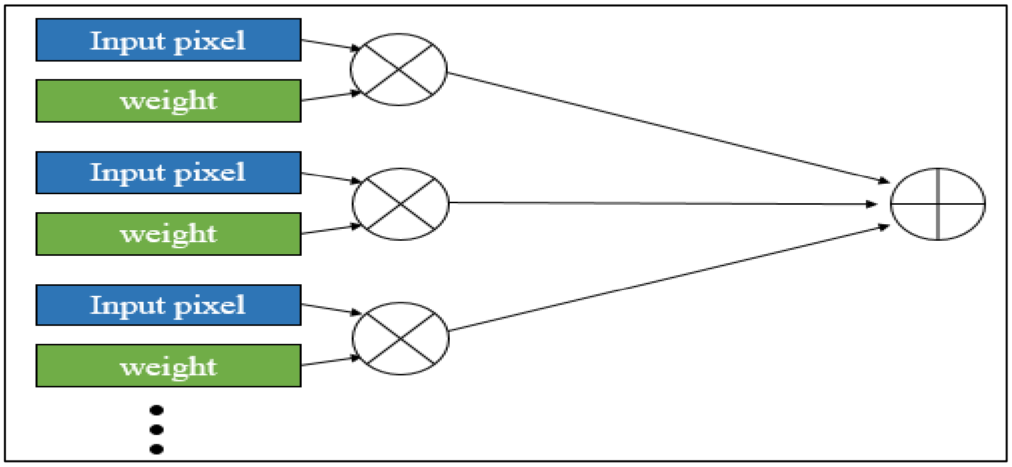


FIGURE 27: INTRA KERNEL PARALLELISM

The challenge against the total parallelism is the available area and resources. Parallelism requires additional modules to perform operations. Now it is obvious the tradeoff between speed and area. Although, parallelism can reduce the time significantly but the limited area and limited memory bandwidth must be taken in to consideration. Choosing a combination of parallelism and pipelining techniques to meet time and area constrains is the main challenge.

1.7 LITERATURE SURVEY

GPUs are known to do well on data parallel computation that exhibits regular parallelism and demands high floating-point compute throughput. Across generations, GPUs offer increased FLOP/s, by incorporating more floating-point units, on-chip RAMs, and higher memory bandwidth. For example, the latest Titan X Pascal offers peak 11 TFLOP/s of 32-bit floating-point throughput, a noticeable improvement from the previous generation Titan X Maxwell that offered 7 TFLOP/s peak throughput. However, GPUs can face challenges from issues, such as divergence, for computation that exhibits irregular parallelism. Further, GPUs support only a fixed set of native data types. So, other custom-defined data types may not be handled efficiently. These challenges may lead to underutilization of hardware resources and unsatisfactory achieved performance.

Meanwhile, FPGAs have advanced significantly in recent years. There are several FPGA trends to consider. First, there are much more on-chip RAMs on next-generation FPGAs. For example, Stratix 10 [16] offers up to ~28 MBs worth of on-chip RAMs (M20Ks). Second, frequency can improve dramatically, enabled by technologies such as HyperFlex. Third, there are many more hard DSPs available. Fourth, off-chip bandwidth will also increase, with the integration of HBM memory technologies. Fifth, these next-generation FPGAs use more advanced process technology (e.g., Stratix 10 uses 14nm Intel technology). Overall, it is expected that Intel Stratix 10 can offer up to 9.2 TFLOP/s of 32bit floating-point performance. This brings FPGAs closer in raw performance to state-of-the-art GPUs. Unlike GPUs, the FPGA fabric architecture was made with extreme customizability in mind, even down to bit-levels. Hence, FPGAs have the opportunity to do increasingly well on the next-generation DNNs as they become more irregular and use custom data types. [17]

The authors of paper [17] consider whether future high-performance FPGAs will outperform GPUs for next-generation DNNs in terms of speed beside its superiority in power consumption-efficiency, evaluating a selection of emerging DNN algorithms on two generations of Intel FPGAs (ArriaTM 10, StratixTM 10) against the latest highest performance Titan X Pascal GPU. They study various general matrix-matrix multiplication (GEMM) operations for next generation DNNs, and then proposed a detailed case study on accelerating Ternary ResNet which relies on sparse GEMM on 2-bit weights (i.e., weights constrained to 0, +1, and -1) and full-precision neurons which its accuracy is within ~1% of the full precision Res-Net which won the 2015 Image-Net competition. The results were very promising; Stratix 10performance is 10%, 50% and 5.4x better in performance (TOP/sec) than Titan X Pascal GPU on GEMM operations for pruned, Int6, and binarized DNNs, respectively. OnTernary-ResNet, the Stratix 10 FPGA is projected to deliver 60% better performance over Titan X Pascal GPU, while being 2.3x better in performance/watt. Results indicate that FPGAs may become the platform of choice for accelerating DNNs.

Authors of [18] from the Institute of Semiconductors from the Chinese Academy of Sciences in Beijing China attempted a small implementation in 2015. Their implementation ran on an Altera Arria V FPGA board operating at 50 MHz. The input images were 32x32, used an 8-bit fixed point data format, and used a 3 Convolution Layers with Activation, 2 Pooling Layers, and 1 Softmax Classifier. This work implemented custom processing elements which could be reconfigured when needed. They measured their performance based on how many images could be processed.

Authors of [19] from the School of Computing, Informatics, and Decision Systems Engineering at Arizona State University. In 2016, one of their research groups afforded to create a scalable FPGA implementation of a Convolutional Neural Network. This group realized that as ever newer Deep Learning CNN configurations increase in layer count, and FPGA implementation would need to keep pace. This group also created a CNN compiler to analyze the input CNN model's structure and sizes, as well as the degree of computing parallelism set by users, to generate and integrate parametrized CNN modules. This implementation ran on a Stratix-V GXA7 FPGA operating with a clock frequency of 100MHz, uses a 16bit fixed data format, consumes 19.5 watts of power, and achieves 114.5 GFLOPS performance. Their FPGA resource utilization is 256 DSPS, 112K Look Up Tables, and 2,330 Block RAM. Their design employs the use of a shared multiplier bank for use with all the multiplication operations.

One of the most limiting hardware realizations for Deep Learning techniques on FPGAs is design size. The trade-off between design configurability and density means that FPGA circuits are often considerably less dense than hard-ware alternatives and so implementing large neural networks has not always been possible. However, as modern FPGAs continue to exploit smaller feature sizes to increase density and incorporate hardened computational units along-side generic FPGA fabric, deep networks have started to be implemented on single FPGA systems. [20]

So not only FPGAs are recommended to be used as hardware accelerators in implementing CNN but also ASIC technologies which gives a better performance, on the other hand the FPGA based accelerators have attracted more attention of researchers than ASIC accelerators because they have advantages of quite good performance, fast development round and capability of reconfiguration.

Authors of [21] consider the spatial architectures used in ASIC and FPGA-based accelerators, discussing how data-flows can increase data reuse from low cost memories in the memory hierarchy to reduce energy consumption. This includes a large global buffer with a size of several hundred kilobytes that connects to DRAM, an inter-PE network that can pass data directly between the ALUs, and a register file (RF) within each processing element (PE) with a size of a few kilobytes or less. They investigate data-flows that exploit three forms of input data reuse (convolutional, feature map and filter). For convolutional reuse, the same input feature map activations and filter weights are used within a given channel, just in different combinations for different weighted sums. For feature map reuse, multiple filters are applied to the same feature map, so the input feature map activations are used multiple times across filters. Finally, for filter reuse, when multiple input feature maps are processed at once (referred to as a batch), the same filter weights are used multiple times across input features maps.

Authors of [22] studied the novel architecture for Squeeze-Net [12] like CNN models and this can be extended to support any CNN model as well. They have addressed two approaches to mitigate resource constraints. First, they use a custom floating point (12 bits for computation and 8bit for storing). The Second is slicing the model into repetitive block called computation blocks. Computation block can be configured dynamically by the host processor to operate in a different mode. They have implemented Squeeze-Net v1.1 for Image-Net for large-scale classification which achieved around 9 FPS at 100MHz. Accuracy loss due to using custom float is measured to be less than 2%. Unlike other implementations which use FPGA boards with a large amount of resources, their experiments are done in DE10-Nano, this mimics actual embedded system like environment.

` While authors of [24] have suggested optimizations to speed up computations in order to efficiently use already trained neural networks on a mobile device. Specifically, they propose an approach for speeding up neural networks by moving computation from software to hardware and by using fixed-point calculations instead of floating-point. They propose a number of methods for neural network architecture design to improve the performance with fixed-point calculations. They also show an example of how existing datasets can be modified and adapted for the recognition task in hand. Finally, they present the design and the implementation of a floating-point gate array-based device to solve the practical problem of real-time handwritten digit classification from mobile camera video feed. The proposed design is successfully implemented in FPGA. In this implementation, input images are processed in real time, and the original image is displayed along with the result. Classification of one image requires about 230 thousand clock cycles and they achieve overall processing speed with the large margin over 150 frames/sec.

Author of [25] from faculty of California State Polytechnic University, Pomona. In spring 2019, he introduces the top-level architectural FPGA implementation involving multiple sub designs communicating on an AXI bus through different sections in [25]. He also discusses how each layer of the convolutional network was accelerated by hardware and software techniques like parallelism in different layers. Alex-Net was implemented on Artix7 XC7A200T, with only 10 classes, not all the 1000 class as the software model, with validation accuracy 0. The frequency is 100 MHz with power equals 1.5 W and latency equals 4062 ms. We compare our work with this work, will be stated in section 7.4.

1.8 SUMMARY

In this chapter a background about convolution neural networks, different architectures, detailed discussion about Squeeze-Net and useful terms such as parallelism and fixed-point representation were presented in addition to literature survey. In next chapter a comparison between 2 different approaches for purposed architecture are discussed.

CHAPTER 2. ARCHITECTURE

In this chapter it’s presented two different architectures for implementing the CNN (Squeeze-Net) on” Virtex7 FPGA”. The first one “Pipelined architecture” that targets high speed implementation of Squeeze-Net by pipelining all the fire modules together and deploying the whole architecture but due to limited resources of the FPGA the number of filters and channels processed at a time per fire module will be small which will not help in achieving the target. The second one “Time-shared architecture” is designinng a very fast 1 block using available FPGA’s resources and reuse the block in different fire modules.

2.1 THE PIPELINED ARCHITECTURE

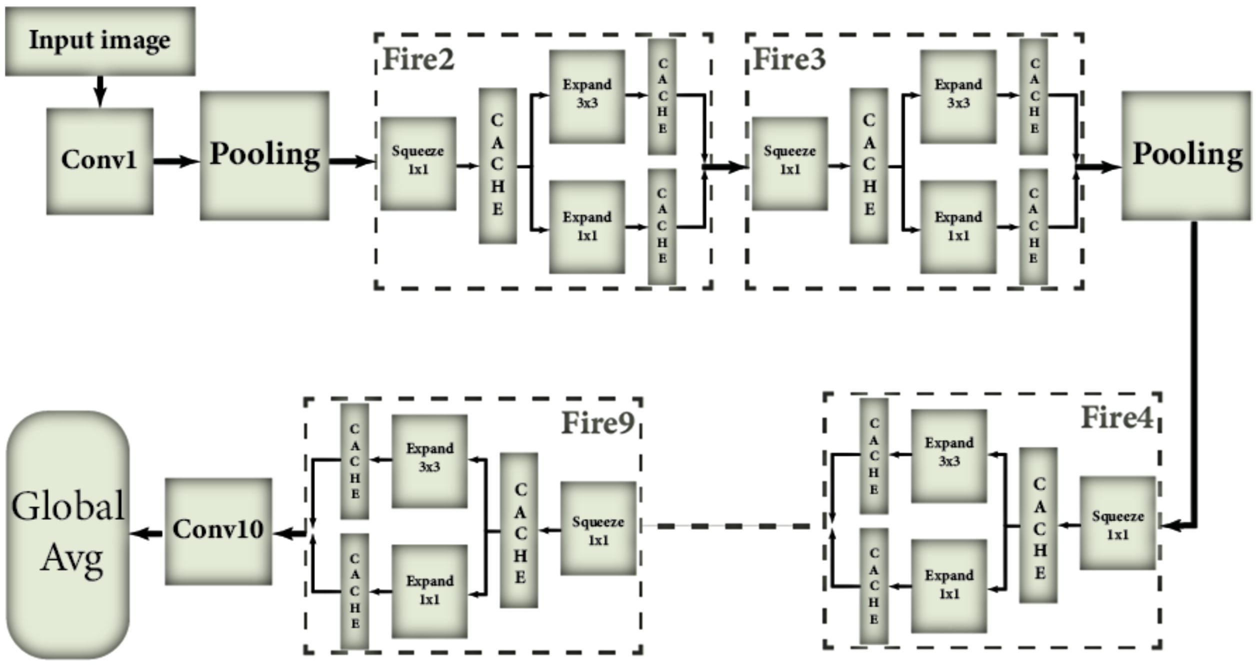


FIGURE 30: OVERVIEW OF PIPELINED ARCHITECTURE

The design metholodgy of this architcture is designing fast CNN by implementing all the 9 fire modules of SqueezeNet on FPGA as shown in figure 30. In each fire module the units needed are:

- Conv 1x1 for squeeze.
- Conv 1x1 for expand.

- Conv 3x3 for expand.

To build this pipliened architecture, in each of the eight fire modules, from fire 1 to fire 9, three “Conv units”,discussed later. Three “Maxpooling” in the whole architecture and one “Conv unit’’ for the first convolution layer (Conv1) and last convolution layer (Conv10) are needed.

2.1.1 CONV UNIT:

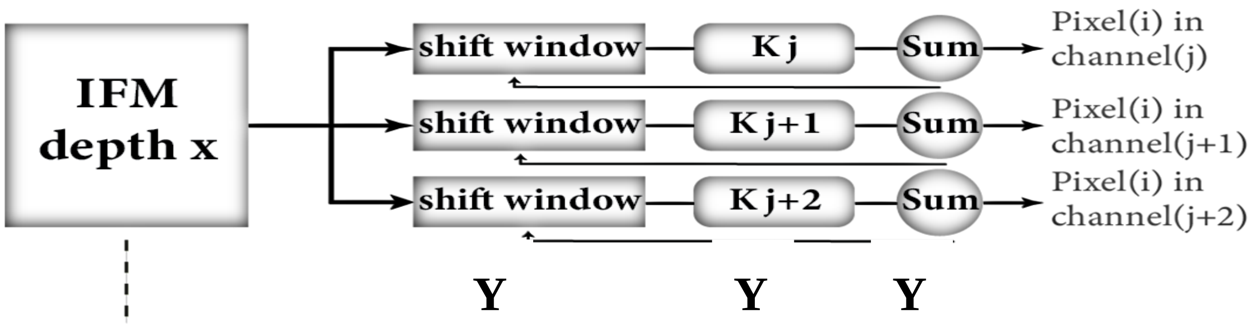


FIGURE 31: CONTROL UNIT OF PIPELINED ARCHITECTURE

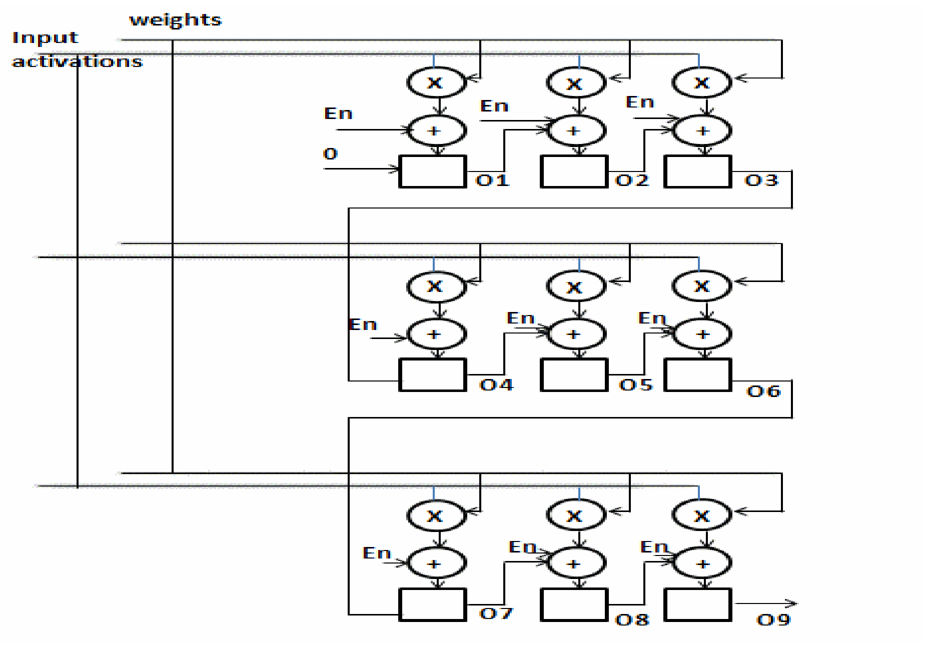


FIGURE 32: K BLOCK IN CONTROL UNIT OF PIPELINED ARCHITECTURE

The Conv Unit, as shown in figure 31, consists of one main block which is K-block (shown in figure 32). Let’s discuss how The Conv Unit works, Firstly the shift window is a convolution shift window that holds a 3 × 3 convolution matrix. K-block describes the Conv Unit, and the weights of the filters saved in this block. The activation inputs (convolution data) received from the shift window after that the convolution operation starts immediately. When this operation has been finished the K-block sends a request to shift the window at this time the Sum will get the result after the accumulation of all depths and bias. The result will be stored in the buffer, and after the memory finishes this reading state, the cache writes the new result back into memory. As discussed in 2.2.1, each fire module needs 3 Conv Units (Expand3x3, Expand1x1and Squeeze1x1). As shown in figure 2, X and Y are design parameters. X is the number of channels which is taken from the IFM (Input Feature Map) as well as from each filter (the depth of the Conv layer). Y is the number of filters processed at a time in each Conv unit. If there is ” n” multiplies of the “X” channels and “m” multiplies of “Y” filters , n times of iterations across the depth to complete one pixel of the OFM (output feature map), by summation the output by SUM block in figure 2 ,and also m times for all filters to complete the depth of OFM. K-block could be pipelined in 3 stages download data, multiply, add. But there will be bubble in Expand 3x3 across the time because every add stage waits the result of the previous one. When a 3x3 window of complete pixels in OFM completed, it can go throw the network and processed by the next layers of the fire modules.

In Expand 1x1 and Squeeze1x1, no adder as 3x3 window of pixels (from O1 to O9) will be generated at a time (no bubbles) but in Expand3x3, 1 pixel will be generated at a time.

To calculate the resources, first assumption for X, Y parameters to be 16, 16 respectively to fasten the convolution operation since ”16” is the greatest common divisor for the number of filters in different layers , the number of DSPs required for the 8 fire modules in Squeeze Net = 16x16x9x8x3 = 55,296 DSPs.

This number of DSPs is too large to be implemented on” FPGA Virtex Ultra Scale”, the used FPGA kit, which contain 2800 DSPs only. We can decrease the number of “X” and “Y” according to the available resources but this will decrease the speed. Therefore, it’s a tradeoff.

2.2 The Time-shared architecture

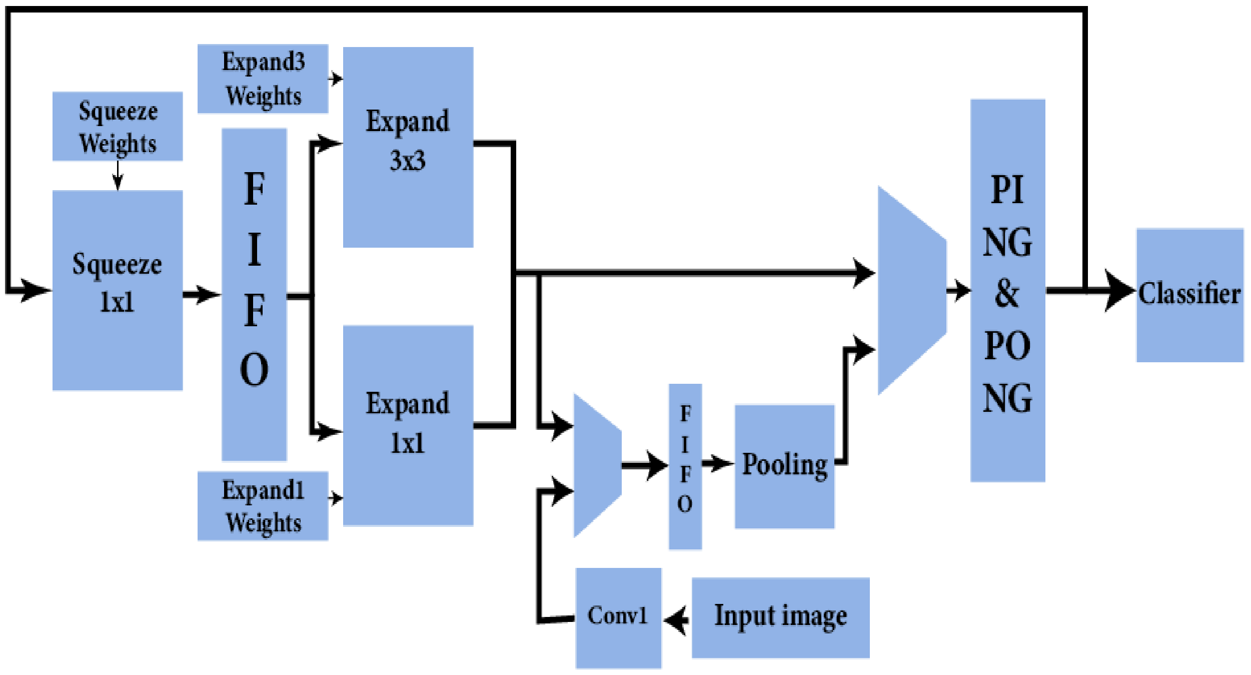


FIGURE 33: OVERVIEW OF TIME-SHARED ARCHITECTURE

The design metholodgy of this architure is designneing a very fast 1 block using available FPGA's resources and reuse the block in different fire modules.The block as shown in figure 33, contains a specific blocks for squeeze layer, expand 1x1 layer and expand 3x3 layer. every block of them is running for specific number of clock cycles depends on the fire module that execautes in this stage. The output of squeeze layer is stored in FIFO before loading as input to Expand layers. The output of expand layers is stored in Data memory and goes to squeeze layer again as input for next stage.

It's useful to remebr that squeeze layers always have a high number of IFM channels (64, 128, 256, 384, 512) and low number of filters(16,32,48,64). On other hand, the expand layer have a low number of input channels but with a high number of filters therefore choosing different method to parallism in each of them is needed.

2.2.1 Squeeze layer

Squeeze layer as shown in figure 34, the block is a 2D matrix array of DSPs. Filters in this layer is only 1x1 filters and the parallism methods were chosen in this layer is Inter output parallelism through channels and filters. In every clock cycle, the block performs operations on a 16 IFM channels with 8 different filters. therefore the latency to claculate the output of 1 pixel in OFM is equal.

$$\text{Latency for 1 pixel} = \frac{\#of\ filters}{8} * \frac{\#of\ Channels}{16}.$$

The total number of DSPs in this layer is 128 DSP.

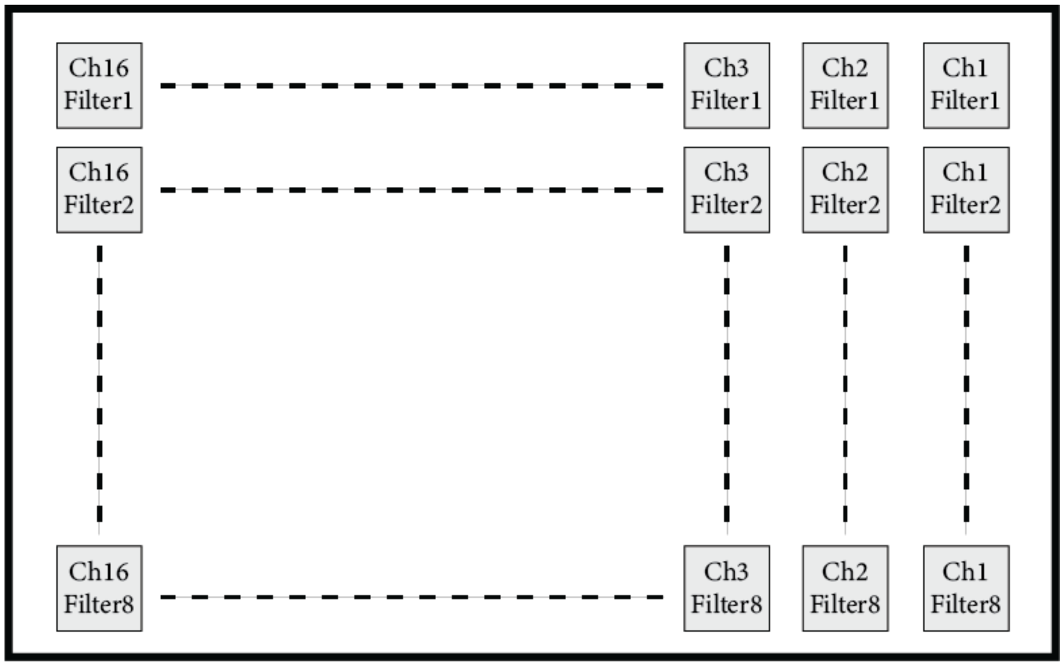


FIGURE 34: ORDERING OF PARAMETERS OF SQUEEZE LAYER

2.2.2 EXPAND LAYER

Expand layer as shown in figure 35, the block is a 2D matrix array of DSPs. Filters in this layer is mix between 1x1 and 3x3 filters and the parallism methods were chosen in this layer are Inter output parallelism through channels and filters and Intra kernel parallelism in Expand 3x3 block. In every clock cycle, the block performs operations on a 16 IFM channels with size 3x3 as shown in figure 36, with 16 different filters and in Exapand 3x3 all 9 multiplications are done in parallel. therefore the latency to claculate the output of 1 pixel in OFM for every block is.

$$\text{Latency for 1 pixel} = \frac{\#of\ filters}{16} * \frac{\#of\ Channels}{8}$$

The total number of DSPs in this layer is 16*8+16*8*9=1280 DSPs.

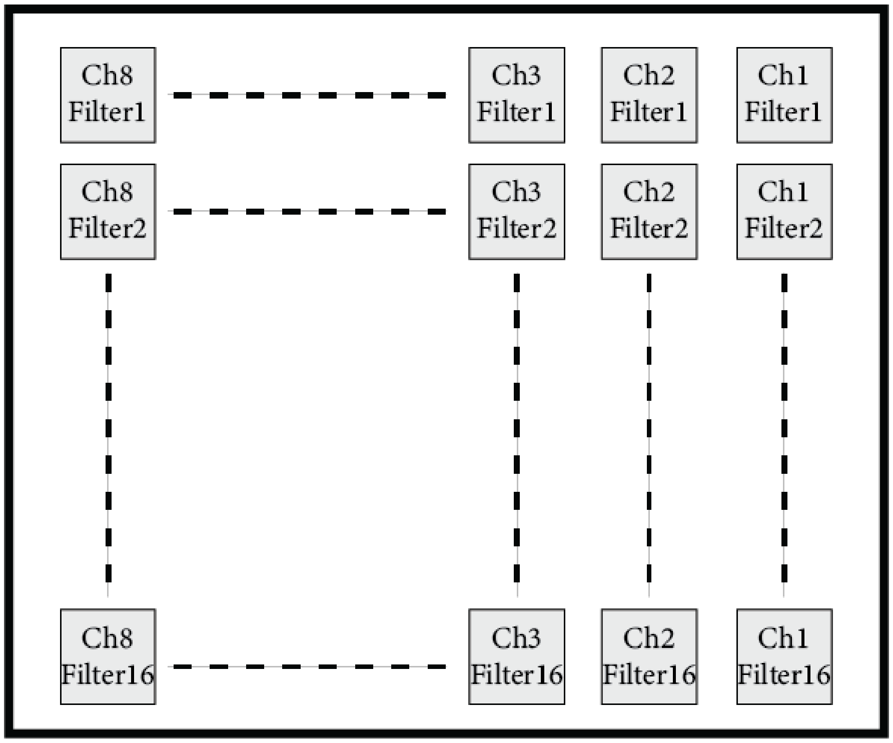


FIGURE 35: ORDERING OF WEIGHTS OF EXPAND LAYER

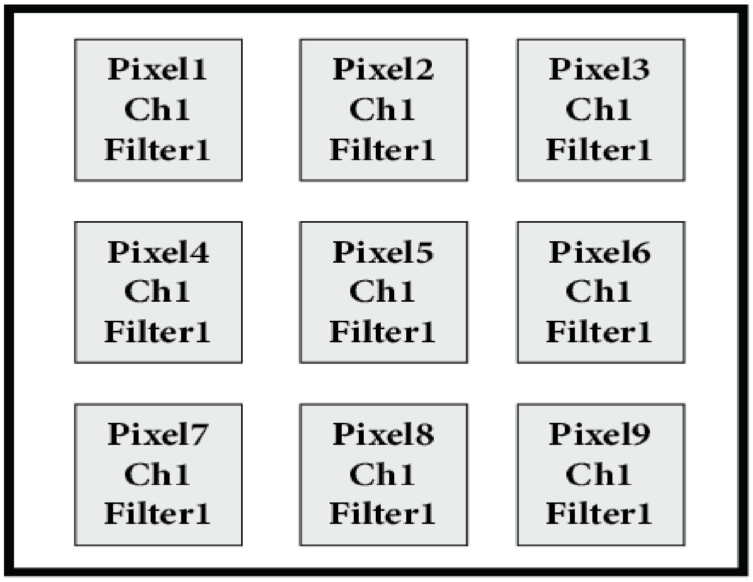


FIGURE 36: INPUT PARAMETERS OF EXPAND LAYER

Table 2: Sizes of IMF and OFM of each Fire stage in SqueezeNet and the estimated number of required clock cycles

Module Num	Layer	input	Output	clk cycles
Fire2	Squeeze	56x56x64	56x56x16	6272
	Expand	56x56x16	56x56x128	12544
Fire3	Squeeze	56x56x128	56x56x16	12544
	Expand	56x56x16	56x56x128	12544
Fire4	Squeeze	28x28x128	28x28x32	6272
	Expand	28x28x32	28x28x256	12544
Fire5	Squeeze	28x28x256	28x28x32	12544
	Expand	28x28x32	28x28x256	12544
Fire6	Squeeze	14x14x256	14x14x48	4704
Fire6	Expand	14x14x48	14x14x384	7056
Fire7	Squeeze	14x14x384	14x14x48	7056
	Expand	14x14x48	14x14x384	7056
Fire8	Squeeze	14x14x384	14x14x64	9408
	Expand	14x14x64	14x14x512	12544
Fire9	Squeeze	14x14x512	14x14x64	12544
	Expand	14x14x64	14x14x512	12544

2.2.3 Adder Tree

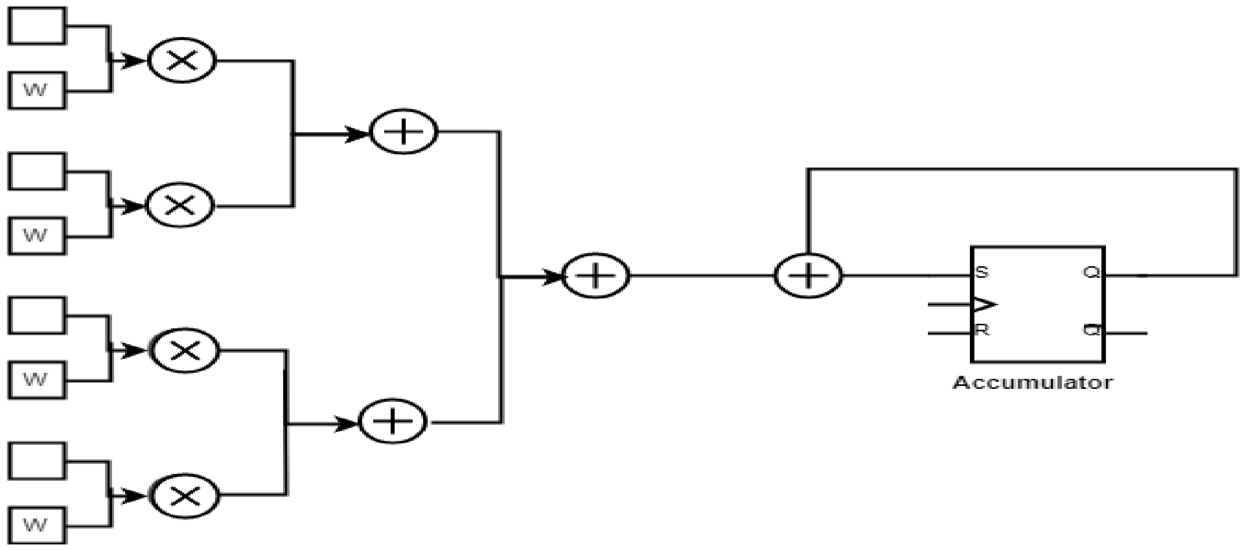


FIGURE 37: ADDER TREE ARCHITECTURE

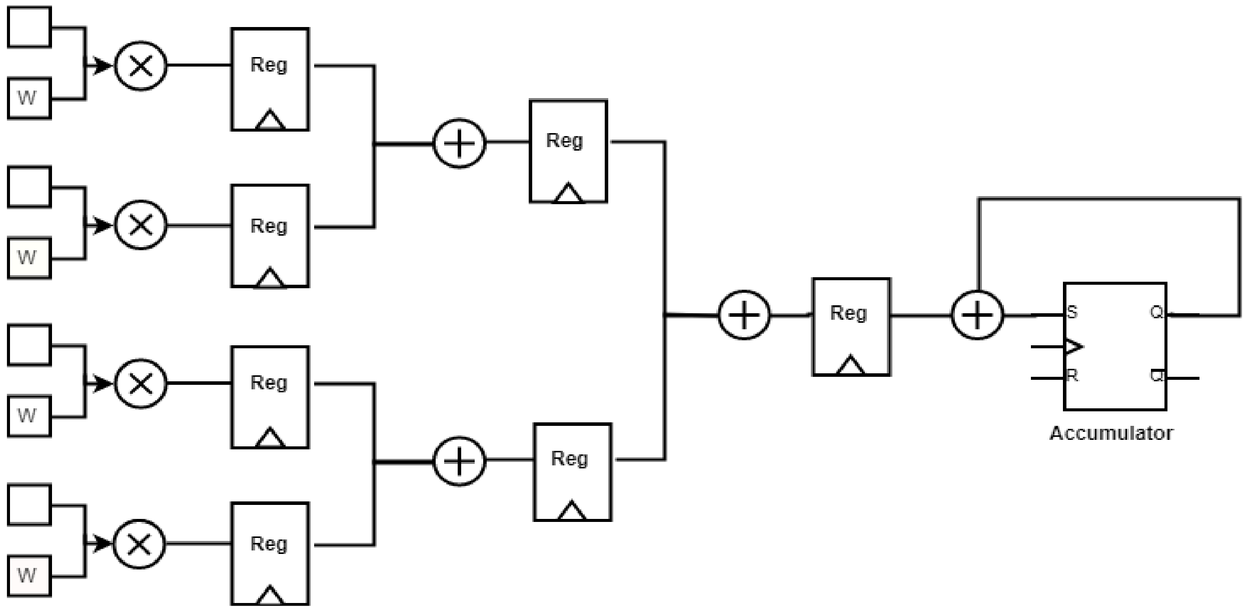


FIGURE 38: PIPELINED ADDER TREE ARCHITECTURE

The adder tree is used to implement 3x3 convolution operation; the adder used is carry save adder as it has 3 inputs. The carry-save adder reduces the addition of 3 numbers to the addition of 2 numbers. The propagation delay is 3 gates regardless of the number of bits. The carry-save unit consists of n full adders, each of which computes a single sum and carries bit based solely on the corresponding bits of the three input numbers.

There are two architectures for the adder tree, the pipelined, shown in figure 38, and the non-pipelined, shown in figure 37. The non-pipelined advantage is the processing of all pixels in one cycle, However it has a long critical path (Multiply + $\log_2(Channels)$ + 1).The pipelined advantages are having short delay path (Multiply) and processing multiple channels simultaneously but it has additional storage area due to the registers.

2.2.4 Data Memory

2.2.4.1 Data distribution

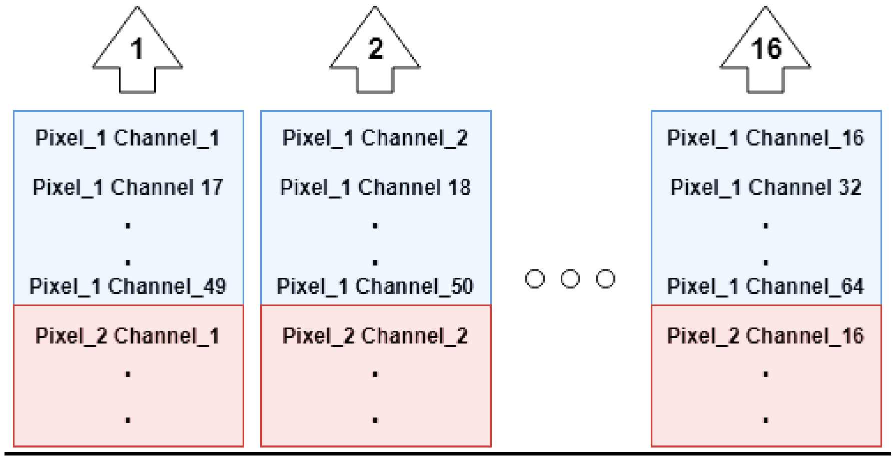


FIGURE 39: DATA DISTRIBUTION IN TIME-SHARED ARCHITECTURE

The data memory is designed to have multiple ports equal to the number of channels fed to the adder tree in one cycle. In this case, the number of ports is 16. In order to achieve this huge number of ports with block rams the data is divided across several block rams as shown in figure 39. Every clock cycle 16 data ports are read from the memory by the squeeze layer.

2.2.4.2 Dual port RAMs

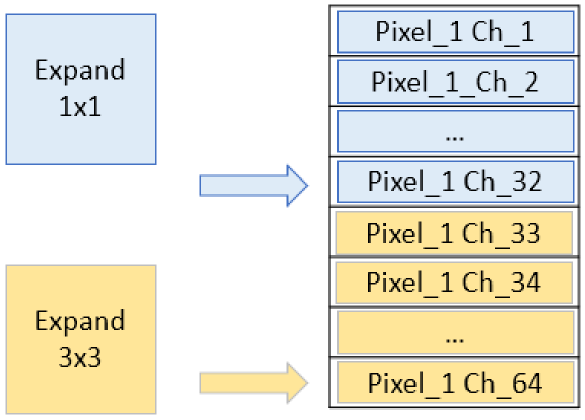


FIGURE 40: DUAL PORT RAM FOE EXPAND LAYERS

The expand stage writes 16 data to the memory in one cycle where 8 data come from expand 1x1 and 8 from expand 3x3. Each block from expand 1x1and expand 3x3 blocks writes in different locations in the memory, as shown in figure 40. Therefore, we use dual port block RAMs where each port is independent from the other.

2.2.4.3 Ping Pong memory structure

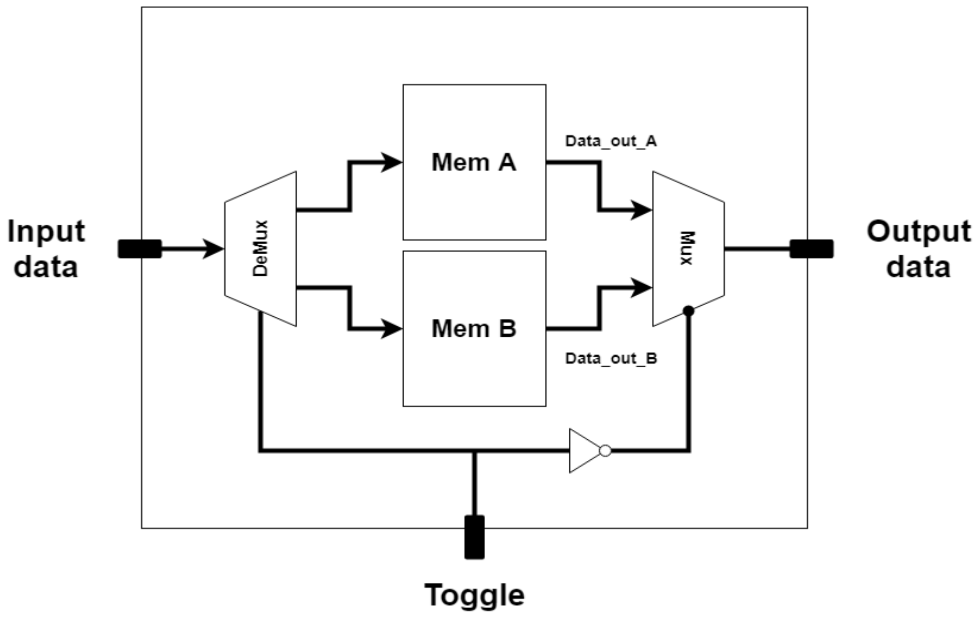


FIGURE 41: PING-PONG MEMORY

When a fire stage starts, the squeeze layer reads data from memory **A** and processes it. After that, the expand layer processes the output of the squeeze layer and stores the data into memory **B**. In the next fire stage, the squeeze layer reads the data from memory **B** while the expand layer stores in memory **A**.

This alternating process can be achieved using ping pong memories where a control signal decides which of the two memories will receive the input data along with the required signals such as the address, write enable, etc. and which memory will output the data. The block diagram is shown in figure 41

2.2.5 WEIGHTS MEMORY

2.2.5.1 DATA DISTRIBUTION

The weights memory has the same distribution as the data memory for one filter weights. In this design 8 filters run in parallel therefore, 8 copies of the same memory are made but with different initialization for each filter.

2.2.5.2 ROMS

Weights are stored in ROMs implemented using either BRAMs or LUTs. The BRAM approach is chosen for layers with lots of weights (expand1x1, expand3x3). The LUTs approach is better for low depth memories so LUTs are used to store weights for the squeeze layer and biases of the whole design.

2.2.6 FIFO BUFFER

The FIFO buffer holds the output data of the squeeze stage and passes it to the expand stage. Once the squeeze finishes processing a pixel, that new pixel with all of its channels gets stored in the buffer. After the buffer gets filled with $W + 2$ entries the expand stage starts processing the data in the buffer.

2.3 SUMMARY

In this chapter 2 different designs are compared with different aspects related to resources. Overviews on the block diagrams of 2 architectures are viewed, one for pipelined architecture that required too much resources and another one for time shared architecture that may needs more clock cycles than the pipelined architecture but with significantly lower number of resources is required. In next chapter the detailed design stages and blocks for time-shared architecture will be discussed.

CHAPTER 3. DETAILED DESIGN

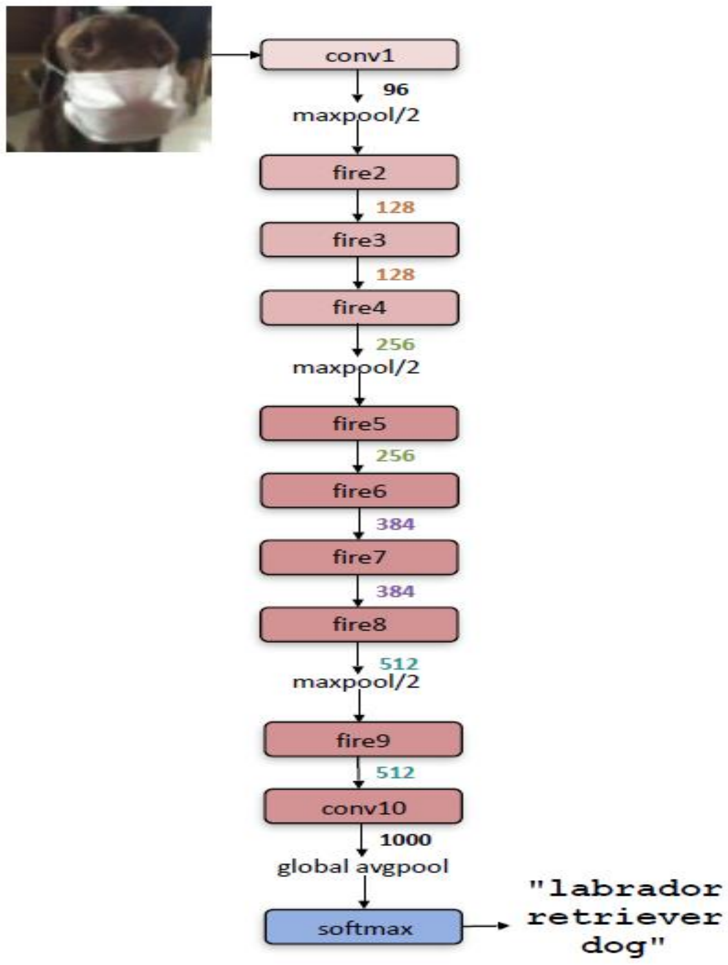


FIGURE 43: SQUEEZENET ARCHITECTURE

As shown in figure 43, we can divide Squeeze-Net architecture into three successive stages as following

Stage1: CONV_1 layer

Stage2: Pooling layer

Stage3: FIRE layer

Stage4: CONV_10 layer.

And each stage reads its filters' weights from weights memory, and its input parameters from image memory as in CONV_1 stage, or from Data memory (PING_PONG memory) as in FIRE and CONV_10 stages. There is another layer that is POOLINHG layer, which is necessary to decrease the number of parameters across the network.

In the following sections, Parallelism of the different stages to speed up the implemented design on FPGA will be described, their inputs and outputs and explain their different methodologies.

3.1 CONV_1 STAGE

CONV_1 layer includes

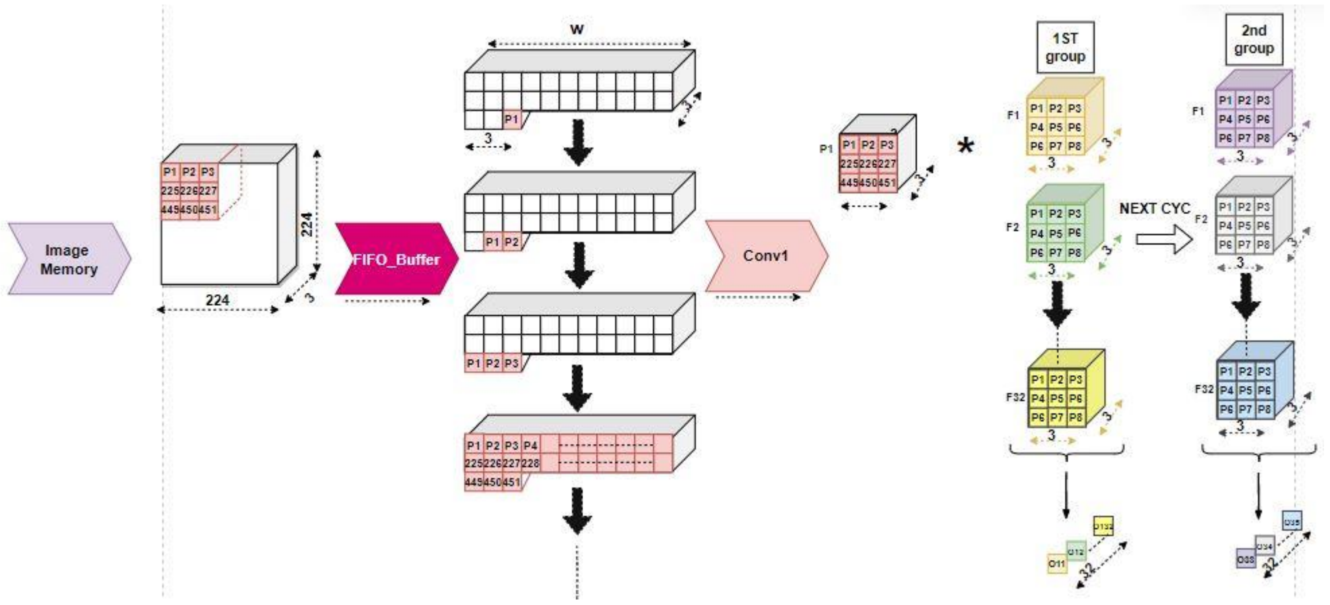


FIGURE 44: CONV_1 STAGE

3.1.1 IMAGE MEMORY

The input image is stored in ROMs implemented using BRAMs approach, as the size of image is large, as shown in figure 44, (224 x 224 x 3) x16 bits =2.3 Mbits.

3.1.2 FIFO BUFFER

FIFO buffer is used before the convolutional layer for real time applications or input videos as only one image can be stored in memory. Therefore, FIFO is needed in this design to ensure synchronization.

SIZE

Size of FIFO = 2*Input_Image_width+3 = 2*224+3,

Where w equals Input_Image_width , with number of channels equals three, as shown in figure 2#,since input image has only three channels (RGB channels) as shown in figure 44.

INPUT

The input of the FIFO buffer is three pixels, one pixel to the tail of each channel of the buffer, from the input image memory and control signals.

OUTPUT

The output of the FIFO buffer is three windows with size 3x3, a window from each channel, to perform 3x3 convolutions.

METHODOLOGY

Initially each register in the FIFO buffer stores ‘0’, while an input pixel is inserted at the tail of the FIFO buffer, the pixel at the head of the FIFO is shifted to the left as shown in figure. After filling the FIFO buffer, convolutional operation is started by taking the first window, after (2w+3) cycles, as shown in figures 45.

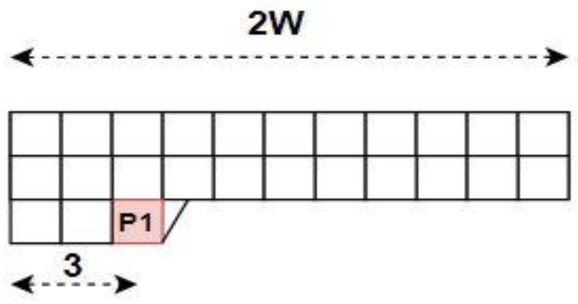


FIGURE 45-A: FIFO BUFFER IN FIRST CYCLE

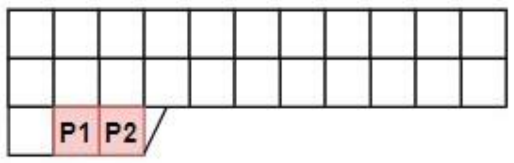
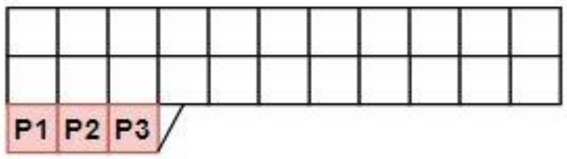


FIGURE 45-B: FIFO BUFFER IN SECOND CYCLE



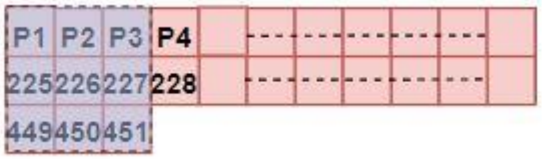


FIGURE 45-C: FIFO BUFFER IN THIRD CYCLE

FIGURE 45-D: FIFO BUFFER AFTER 2*W+3 CYCLES

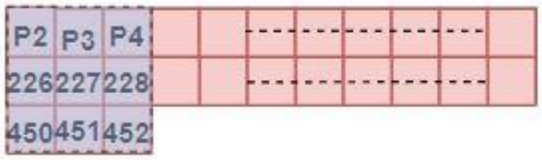


FIGURE 45-E: FIFO BUFFER AFTER 2*W+4 CYCLES

3.1.3 CONVOLUTIONAL LAYER

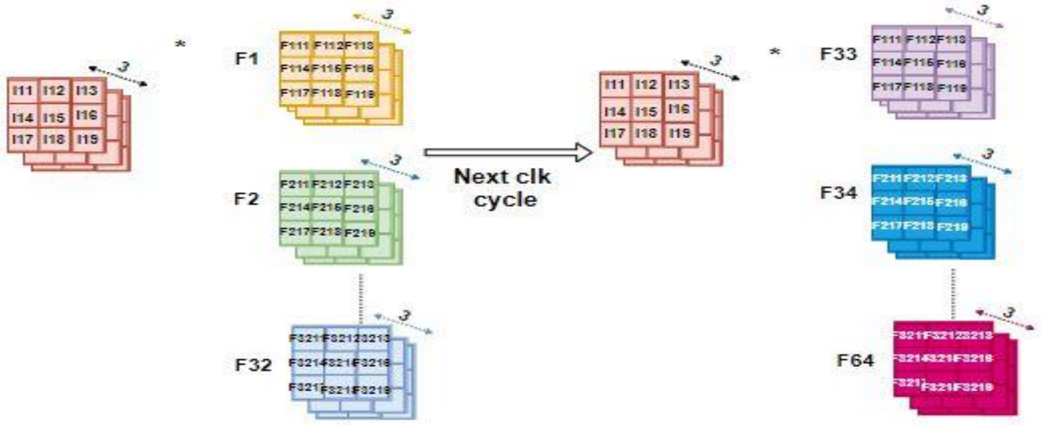


FIGURE 46: PARALLELISM OF CONVOLUTIONAL LAYER IN CONV_1 STAGE

This is the core of CONV_1 stage. In this layer there are 64 different filters with size 3x3, with stride equals 2 and number of channels equals 3, since the depth of the filters equals to the depth of the input feature map, (the depth of the input image), as shown in figure 44.

PARALLELISM

Since filters have depth of three channels which is small number of channels, then convolution is performed to the whole depth. And to speed up the convolution operation, 32 filters are executed out of 64 filters in one cycle for a window of parameters then in the next cycle the other 32 filter will be convolved with the same window of parameters, as shown in figure46.

INPUT

The inputs of the convolution layer are

1. 32x9x3 pixels represent the 32 filters with size 3x3 across depth of eight channels from weights memory.
2. 32 pixels represent the filter biases.
3. 9x3 pixels represent the input image parameters from the FIFO buffer.
4. Control signals.

OUTPUT

The output of the convolutional layer is 32 pixel in depth in output feature map, as shown in figure #, and then propagates through the next layer which is POOLING layer.

METHODOLOGY

The input compound window (a window from input feature map with its depth) from the buffer enters this layer and the operation is held by convolving it with 32 filter in parallel as shown in figure3#. Let’s describe the operation in details by taking the first window of the input, with its depth, with one filter as shown in figure 47.

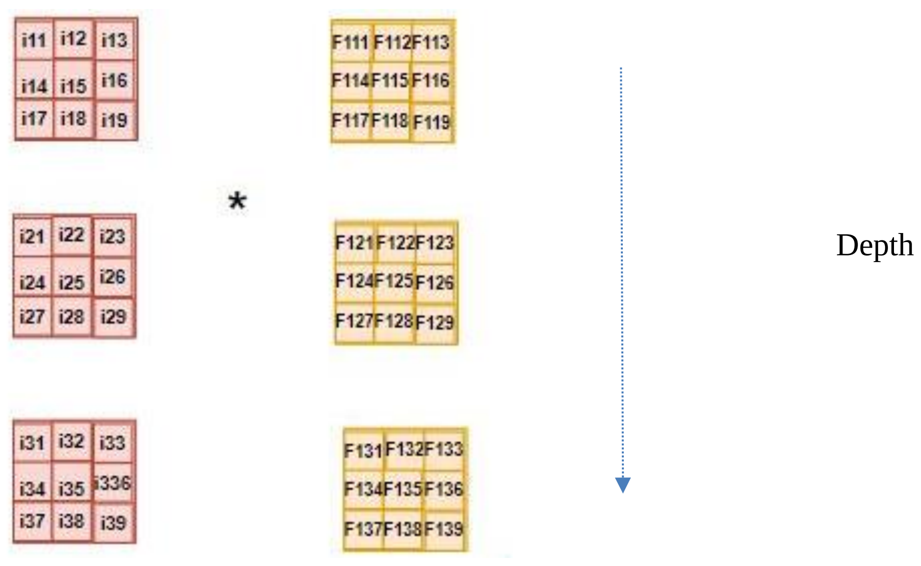


FIGURE 47: CONVOLUTION OF THE WINDOW OF IFM WITH FILTER OF DEPTH =3

The convolution operation is executed by multiplying each parameter pixel to the corresponding pixel in the filter by using DSPs, adding the product of each window among the depth, 3 channels, together. Then adding the result of the 3 channels together, as shown in figure 47. By using MAC operation of the last DSP module, that multiplies the 9th parameter with the 9th weight of the last channel, the bias of each filter has been added. This technique decreases the number of the used LUTs as no extra adder is needed, as shown in figure48.

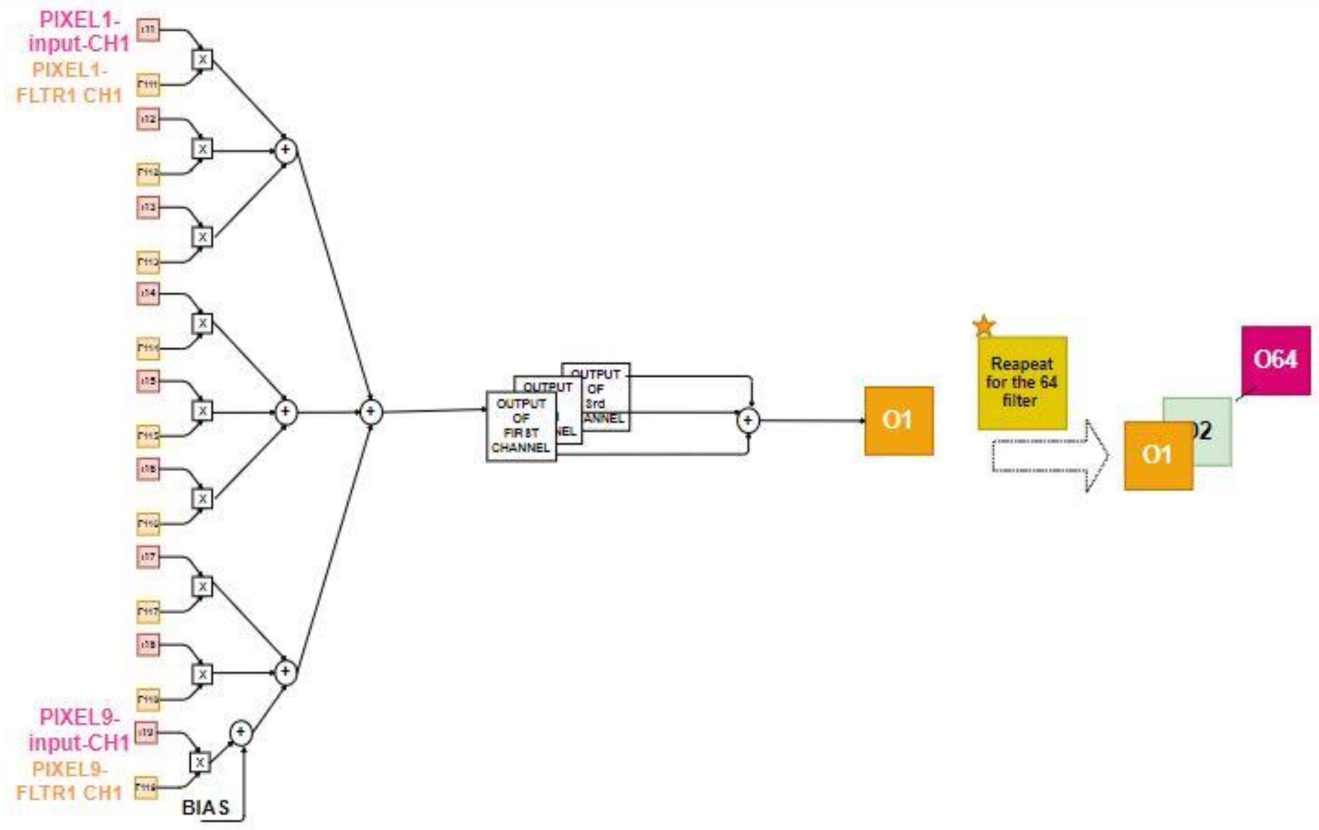


FIGURE 48: CONVOLUTION OPERATION OF THE FIRST FILTER WITH A WINDOW OF IPM

Every filter represents a channel in the output feature map as discussed in chapter 1, Therefore the final OPM depth is 64, as shown in figure 48.

For convolving with stride 2, horizontal stride 2. After filling the FIFO buffer, a counter starts initially with ‘0’; the first window is propagated through the following layer while the next window is ignored. Therefore, each even window is propagated to convolutional operation while the odd one is ignored, as shown in figure 49 . While for vertical stride 2 all windows whose counter number is between (FIFO_WIDTH -2) to (2* FIFO_WIDTH -1) are ignored to skip the second row.

Zero-Padding is used in this stage because filter size is 3x3 and input image size 224x224 which results in the filter window slides out of the input image boarders as in figure 48. Therefore, Zero-Padding is done at the left and the bottom boundaries for validation of convolution operation

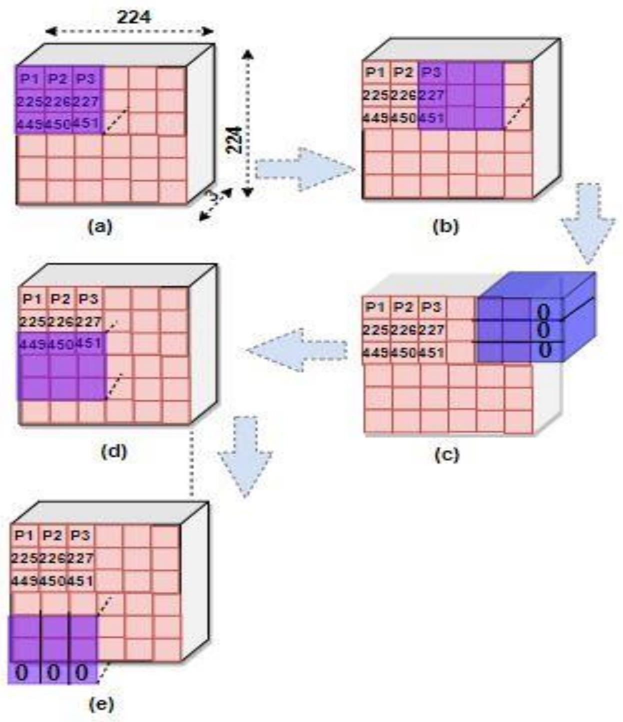


FIGURE 48: STRIDE 2 OF CONVOLUTION LAYER I CONV_1 STAGE WITH ZERO-PADDING

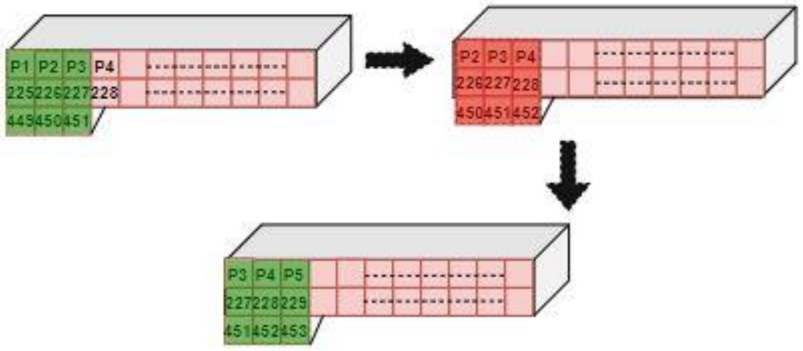


FIGURE 49: THE WINDOW OF EVEN COUNT PROPAGATES (GREEN), WHILE THE ODD ONE IS IGNORED (RED)

3.2 MAX-POOLING LAYER

In Squeeze-Net version1.1, Max-Pooling layer is used after three layers throughout the network, which are CONV_1 layer, FIRE_3 and FIRE_5. Since we use a time-shared design, then this layer is common after those three layers and also since the Max-Pooling needs a window of parameters of size equals 3x3 and stride equals 2, then a FIFO buffer is used before Max-Pooling layer.

3.2.1 FIFO BUFFER

FIFO buffer is used before Pooling layer to get a window with size 3x3 simultaneously to output the maximum pixel among this window.

SIZE

Since POOLING layer is used after CONV_1, FIRE_3 and FIRE_5, which have different sizes and different number of channels, then the size of FIFO buffer is varying across its channels for full utilization.

- CONV_1 output feature map (OFM) have 64 channels with size 112x112.
- FIRE_3 OFM have 128 channels with size 55x55.
- FIRE_5 OFM have 256 channels with size 28x28.

Then the number of channels in this FIFO buffer is 256 channels with the size 2*W+3. We divide the depth into 8 sets as shown in figure 50, 32 channels for each as output of the pervious stage (CONV_1, FIRE_3 or FIRE_5) is 32 pixels in depth.

For the first 2 sets W equals 112 for OFM of CONV_1 size as number of channels of this stage equal 64.Although every fire has different size and depth, Therefore handling these different sizes throughout the design flow is done by control signals .For example, in the next 2 sets W equals 56 according to the size OFM of FIRE_3 that is 56x56x128, and for the last 4 sets channels W equals 28 since size of OFM of FIRE_5 is 28x28x256.

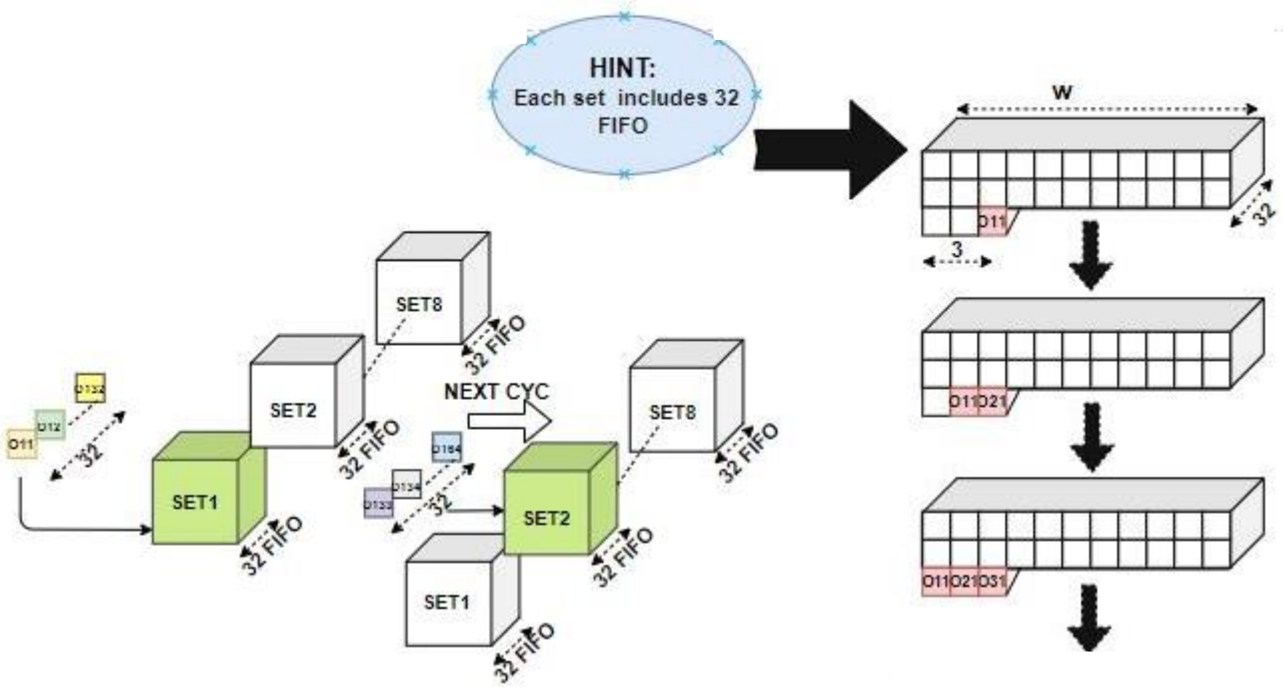


FIGURE 50: THE FIFO BUFFER SETS BEFORE POOLING LAYER

INPUT

The input of the FIFO buffer is 32 pixels, one pixel to the tail of each channel of the buffer, from the input image memory and control signals.

OUTPUT

The output of the FIFO buffer is 32 windows with size 3x3, a window from each channel, to perform 3x3 convolutions.

METHODOLOGY

Initially each register in the FIFO buffer stores ‘0’, while an input pixel is inserted at the tail of the FIFO buffer, the pixel at the head of the FIFO is shifted to the left as shown in figure50. After filling the FIFO buffer, the window is propagated to the next layer as discussed in section 3.1.

3.2.2 Pooling layer

The pooling function is to reduce the number of parameters after certain convolution layers to classify the input image to small number of classes in the last layer. The input of any pooling layer in SqueezeNet is a window with size 3x3x32 pixels and the output is the maximum pixel, as shown in figure 9#.

Parallelism

Since in CONV_1 layer, FIRE_3 and FIRE_5, 32 filters are executed in parallel for the same window of parameters in 1 cycle and the pooling layer is following these layers, then we design the pooling layer to take 32 windows for 32 channels in parallel in one cycle.

Input

The inputs of the pooling layer are

1. 32x9 pixels represent the 32 channels with size 3x3 from FIFO buffer.
2. Control signals.

Output

The output of the convolutional layer is 32 pixels for different channels in output feature map and stored in PING-PONG memory.

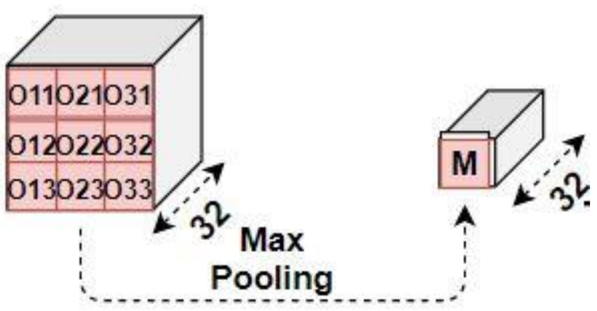


FIGURE 51: THE POOLING LAYER

METHODOLOGY

Pooling layer is responsible for getting the important information from IFM to decrease number of parameters across network. The type of Pooling used in SqueezeNet is 'Max-pooling', at which the most important information among the input window is the maximum parameter. Flow chart of the Pooling layer is shown in the following figure 52. Simply we get the maximum of each row by comparing its parameters, using behavioral comparator. Then we compare the maximum pixel of each row to get the maximum pixel of the whole window.

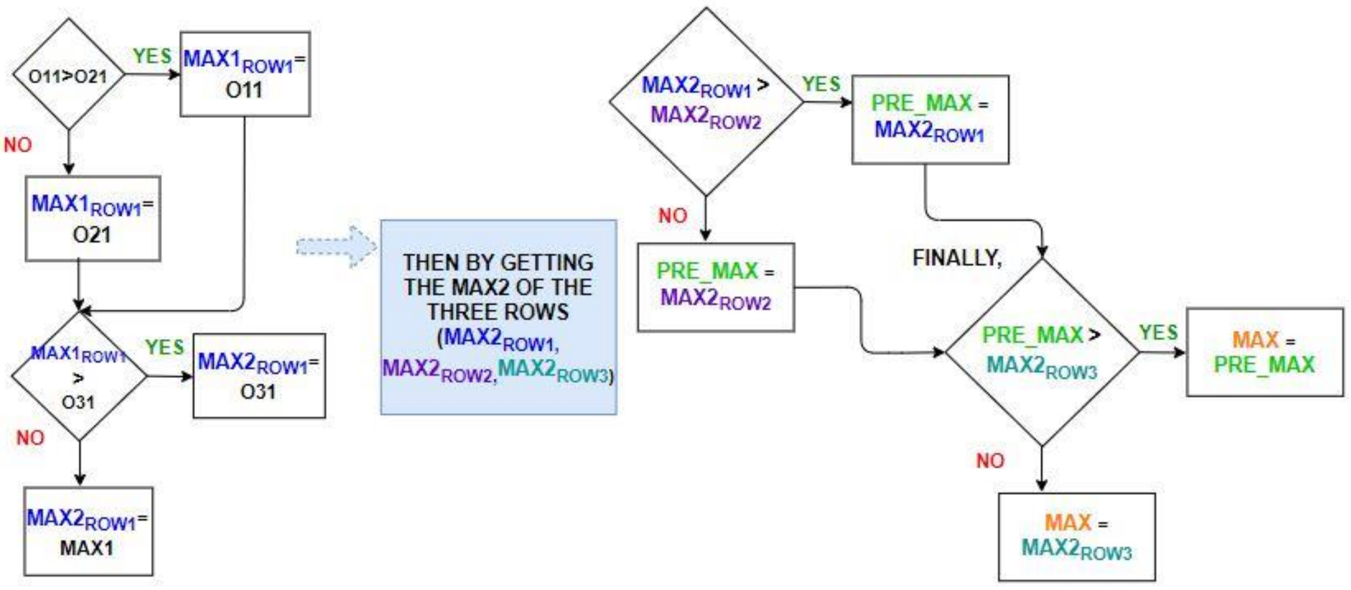


FIGURE 52: FLOW CHART OF POOLING LAYER

3.3 FIRE STAGE

Each Fire Module in Squeeze-Net divide into 2 main layers, which are SQUEEZE layer and EXPAND layer and PING-PONG Memory as discussed in chapter1.

3.3.1 PING-PONG MEMORY

This Memory follows FIRE_2, FIRE_4, FIRE_6, FIRE_7, FIRE_8 and Pooling layer.

INPUT

There are 2 input ports for Ping-Pong memory; each port has size of 16 pixels, 16x16 Bits, addresses and control signals.

OUTPUT

The output of the Ping-Pong memory is one output port with size 16x16 Bits.

METHODOLOGY

The Size of input port (2x16x16 Bits) is determined as the output of Conv_1 layer in one cycle is 32 pixels in depth, as discussed in section 3.1, then they are stored in two consecutive addresses, as shown in 53.There are 2 different input ports, each of size 16x16, because the output of Expand layer which will be discussed later on in this section, is 16 pixels from Expand 1x1 and another 16 pixels from Expand 3x3. Each 16 pixels are stored in apart addresses as OFM of Expand 3x3 module is concatenated with OFM of that of Expand 1x1, as shown in 53.

As discussed in section2.2.4, the memory that receive the input data has toggled with the memory that outputs the data after each Fire module due to pipelined design that allows reading and writing data in different layers simultaneously, as shown in figure 53.

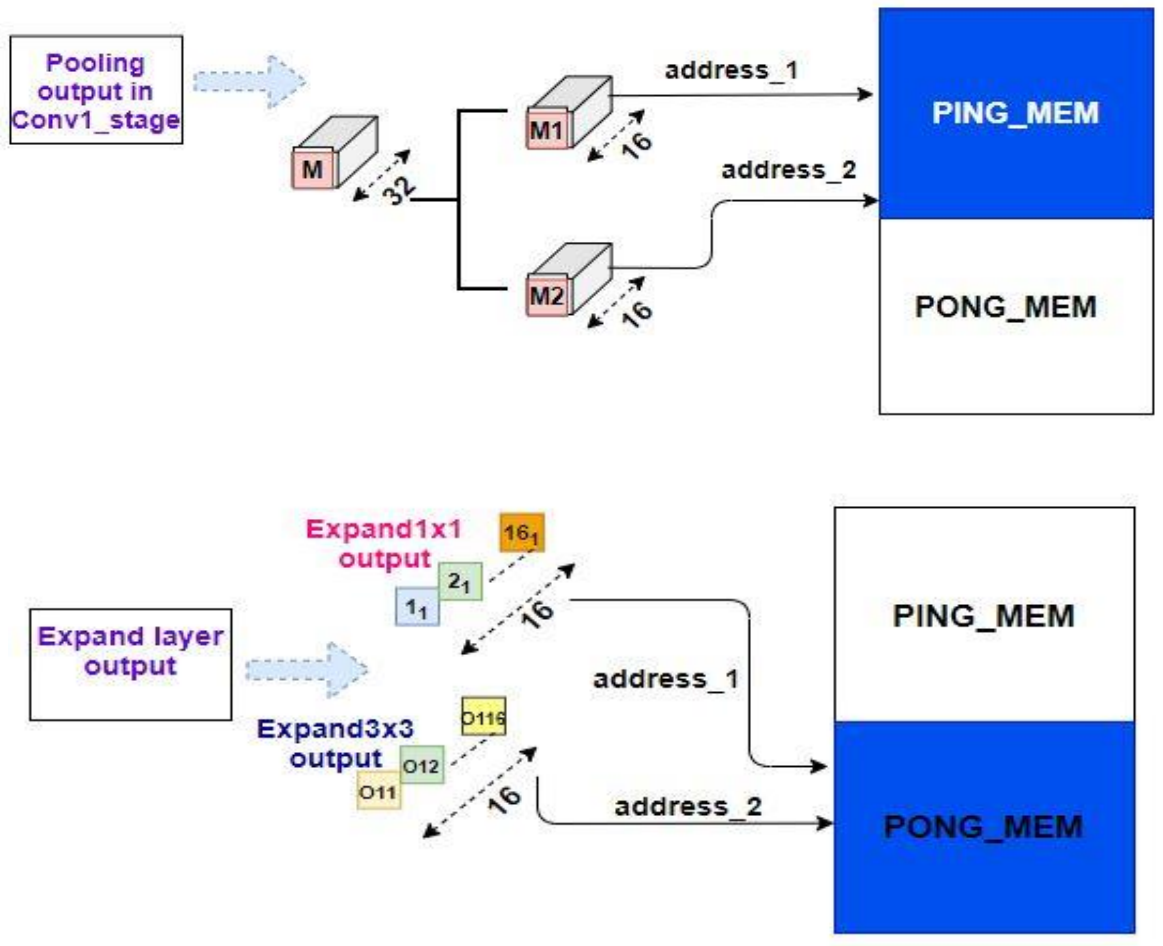


FIGURE 53: TOGGLE OF PING-PONG MEMORY ACROSS NETWORK

3.3.2 Squeeze layer

This layer is a convolution layer of filters with size 1x1 and number of filters relatively small compared to next layer (Expand layer).

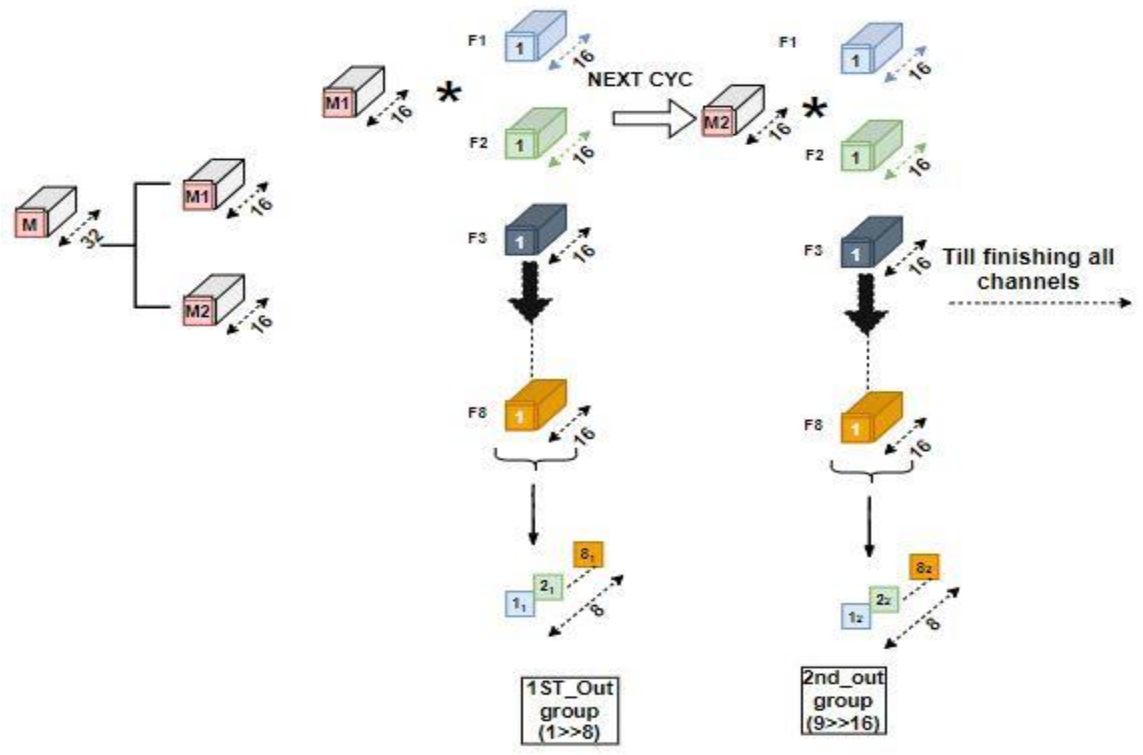


FIGURE 54: INPUT AND OUTPUT OF SQUEEZE LAYER

Parallelism

Since input feature map (IFM) of this stage has large number of channels, starts with 64 channels ,the output of CONV_1 stage, and ends with 512 channels, the output of FIRE_8 module. In addition to small number of filters in Squeeze layer, starts from 16 filters in FIRE_1 module and ends with 64 filters in FIRE_9 module, then we design Squeeze layer to convolve 8 filters with 16 channels with the same window of 16 channels from IFM in parallel in one clock cycle.

Input

The inputs of the Squeeze layer are

1. 8x16 pixels represent the 8 filters with size 1x1 across depth of 16 channels from weights memory.
2. 8 pixels represent the filter biases.

3. 8 pixels represent the IFM parameters from the Ping-Pong memory.
4. Control signals.

Output

The output of the Squeeze layer is 1x1x8 pixels in OFM after number of clock cycles = number of channels in Squeeze module / 16 (number of parallel channels), and stored in the following FIFO buffer.

As shown in figure 54, a compound pixel is completed after number of cycles equals the number of IFM channels in this layer /16, where the 32 channels pixel input is the output of conv1 and expand module in any fire at one cycle that is saved in Ping-Pong memory.

Methodology

In one cycle we convolve 16 channels of IFM with the opposite 16 channels of 8 filters, and then accumulate the result of this convolution until we finish all the depth (number of channels) of the same window. Subsequently we repeat this operation until finishing all the filters, and then we can switch to the next window, the operation is declared in only one window as shown in figure.

Hint: Accumulation on channels and Repeat for filters. Bias is added for each filter as shown in figure 55.

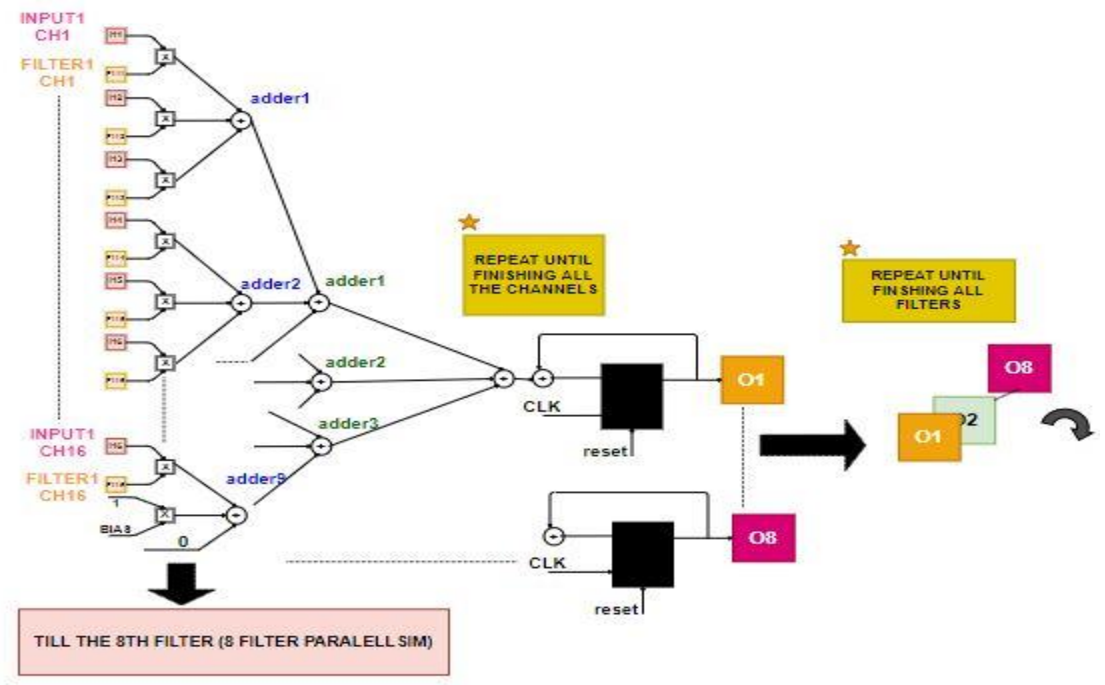

INPUT1 CH1
FILTER1 CH1
adder1
adder2
adder1
adder2
adder3
adder8
INPUT1 CH16
FILTER1 CH16
BIAS
0
REPEAT UNTIL FINISHING ALL THE CHANNELS
REPEAT UNTIL FINISHING ALL FILTERS
CLK
reset
O1
O8
CLK
reset
O8
O2
O1
TILL THE 8TH FILTER (8 FILTER PARALELL SIM)

FIGURE 55: CONVOLUTION OPERATION IN SQUEEZE LAYER

3.3.3 FIFO BUFFER

As discussed in previous sections a FIFO buffer is used when the convolution is of size 3x3. Same design of FIFO buffer is used but with size 2*W+3, where W is the size of IFM that is 55x55. Number of channels of this FIFO buffer equals 64 which is the maximum number of filters in the squeeze layer of FIRE_9, at which number of channels are divided into 8 sets each of 8 channels because the output of Squeeze layer is 8 channels, as declared in figure 56.

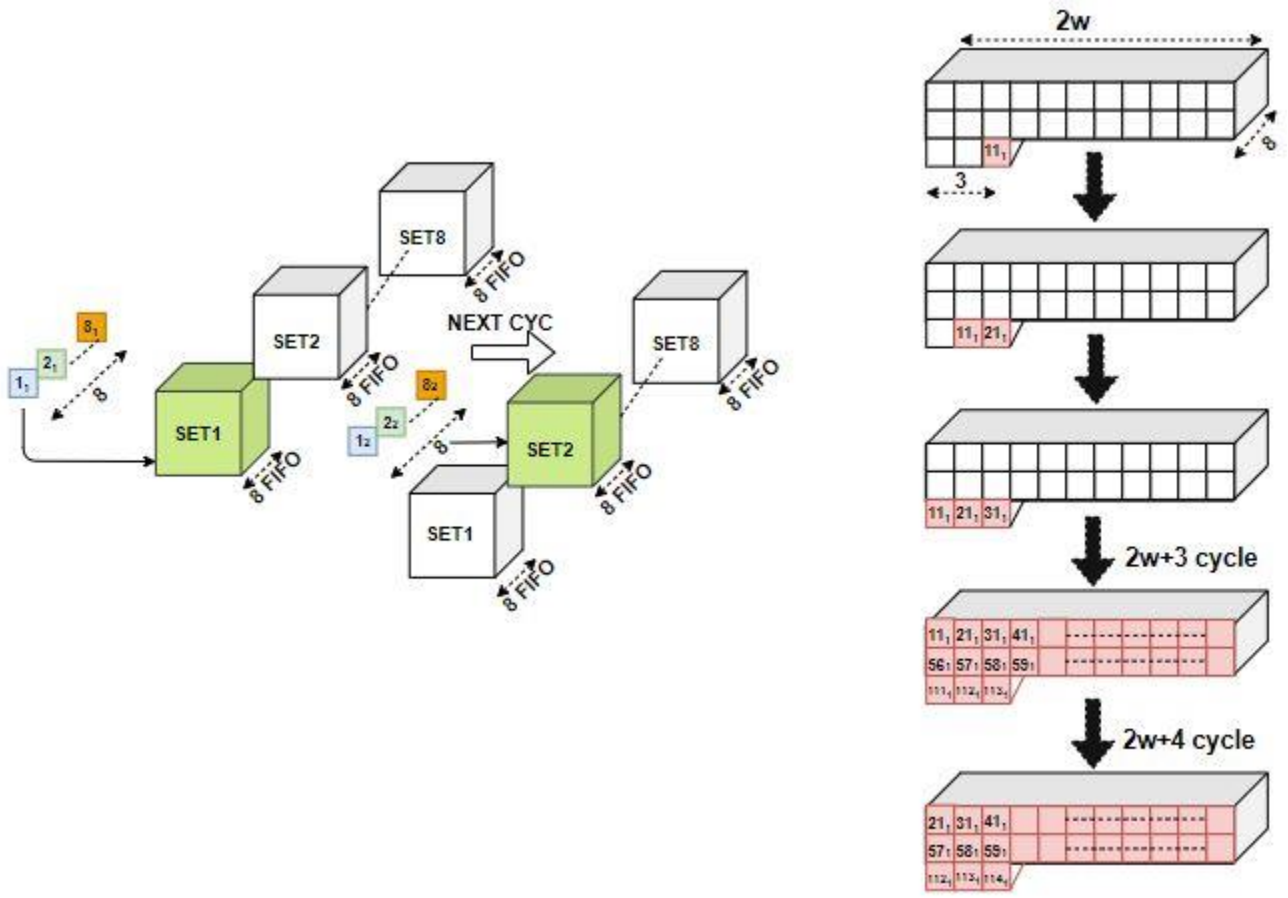


FIGURE 56: FIFO BUFFER BETWEEN SQUEEZE LAYER AND EXPAND LAYER

3.3.4 EXPAND LAYER

This layer is divided into 2 layers which are Expand_1x1 where filters of size 1x1 and Expand_3x3 where filters of size 3x3.

PARALLELISM

Since input feature map (IFM) of this stage has small number of channels, starts with 16 channels ,the output of FIRE_2 stage, and ends with 64 channels, the output of FIRE_9 module. In addition to large number of filters in Expand layer, starts from 128 filters in FIRE_1, 64 filters in Expand_3x3 and same in Expand_1x1, and ends with 512 filters in FIRE_9 64 filters in Expand_3x3 and same in Expand_1x1. Then we design Expand layer to convolve 16 filters with 8 channels with same window of 8 channels from IFM in parallel in one clock cycle.

INPUT

The inputs of the Expand layer are

1. 16x8 pixels represent the 16 filters with size 1x1 across depth of 8 channels for Expand_1x1 from weights memory.
2. 16x8x9 pixels represent the 16 filters with size 3x3 across depth of 8 channels for Expand_3x3 from weights memory.
3. 16 pixels represent the filter biases.
4. 16x9 pixels represent the IFM parameters from FIFO buffer.
5. Control signals.

OUTPUT

The output of the Expand layer is 1x1x16 pixels in OFM of Expand 1x1 and 1x1x16 pixels in OFM of Expand 3x3 completed after number of cycles equals to number of channels in Expand module / 8 (number of parallel channels), and stored in the FIFO buffer in case of FIRE_3 and FIRE_5 or stored in Ping-Pong memory in any other fire module which is determined by control signals.

METHODOLOGY

In one cycle we convolve 8 channels of IFM with the opposite 8 channels of 16 filters, and then accumulate the result of this convolution until we finish all the depth (number of channels) of the same window in OFM. Subsequently we repeat this operation until finishing all the filters, then we can switch to the next window, as shown in figure 58.

The result of Expand_3x3 is concatenated after result of Expand_1x1 to get the final OFM of the Fire module, then stored again in Ping-Pong memory or passes to pooling layer after certain fires (Fire_3 and Fire_5),as shown in figure 57. .

Also figure 57 declared that each window with 8 channels is convolved with 16 filters from Expand 3x3 layer and the center pixel is taken to be convolved with 16 filters from Expand 1x1 layer at the same time.

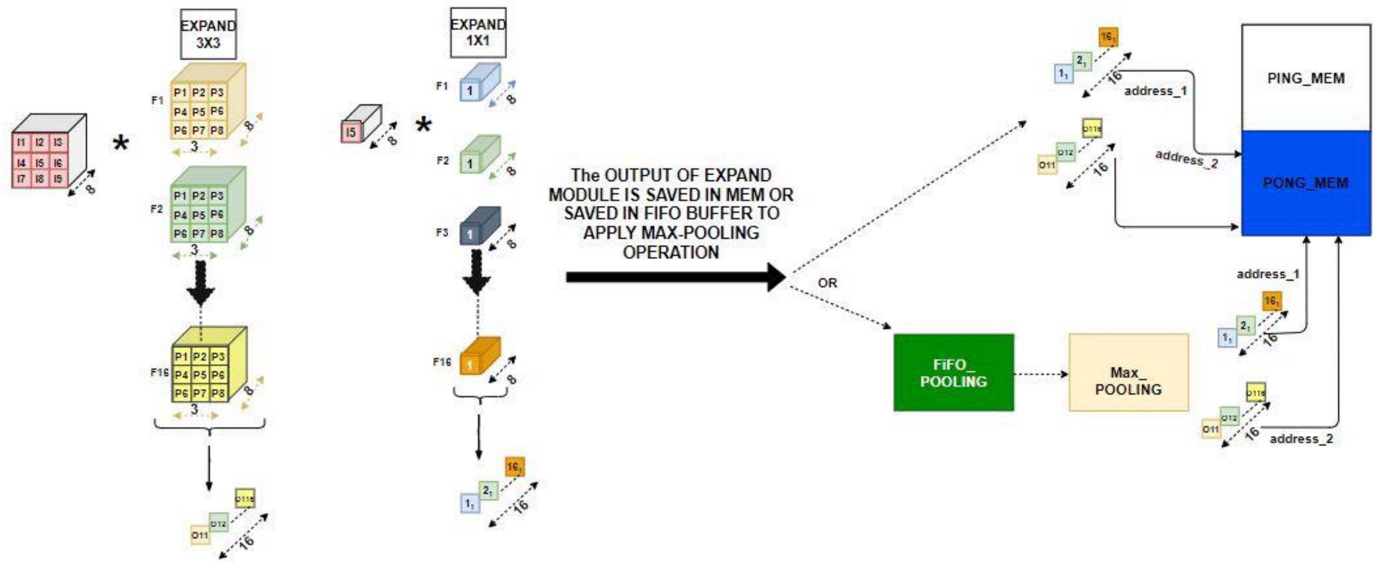

EXPAND 3X3
EXPAND 1X1
F1
F2
F16
F3
*
The OUTPUT OF EXPAND MODULE IS SAVED IN MEM OR SAVED IN FIFO BUFFER TO APPLY MAX-POOLING OPERATION
OR
PING_MEM
PONG_MEM
address_1
address_2
FIFO_ POOLING
Max_ POOLING
16
8
3

FIGURE 57: INPUT AND OUTPUT OF EXPAND LAYER

In Expand3x3 module, as shown in figure 58, 3x3x8 pixels of an IFM window convolves with 3x3x8 pixels of 16 different expand_3x3 filters in parallel at one cycle. Then accumulate the result of this convolution until we finish all the depth (number of channels) of the same window. Subsequently we repeat this operation until finishing all the filters, and then we can switch to the next window, the operation is declared in figure 16#. Bias is added for each expand_3x3 filters in Expand_3x3 module as shown in figure 58.

In Expand1x1 module that shown in figure 15#, is the same as expand 3x3 as the middle pixel, (i5 in figure 57), convolves with 16 different expand_1x1 filters in parallel at one cycle, as shown in figure 57.Then accumulate the result also till finishing all the channels of IPM and repeating this operation till finishing all the expand_1x1 filters. Bias is added for each expand_1x1 filters in Expand_1x1 module as shown in figure 58.

Zero-Padding is used in this stage with type ‘Same’ because the size of filters in Expand_3x3 is 3x3 and the size of IFM is 56x56, which resulted in OFM with smaller size, as discussed in chapter 1 in section 1.2.1, and cannot be concatenated with the OFM of Expand_1x1 which has the same size of IFM. Therefore, Zero-Padding is done at all boundaries of IFM by starting convolution operation after filling the FIFO buffer before Expand layer with only W+2 pixels for padding the upper boundary. This is also the reason why we take the center pixel for Expand_1x1 convolution, as shown in figure 60.

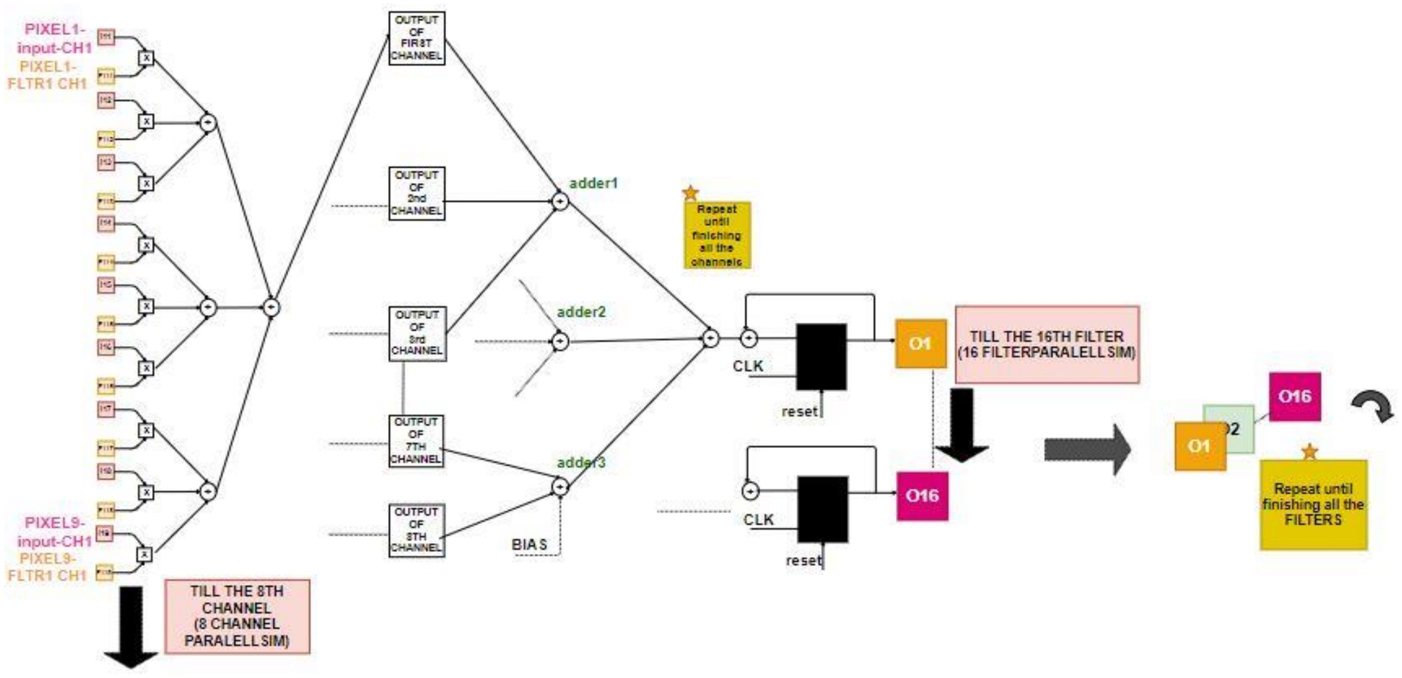


FIGURE 58: CONVOLUTION OPERATION IN EXPAND_3X3 MODULE

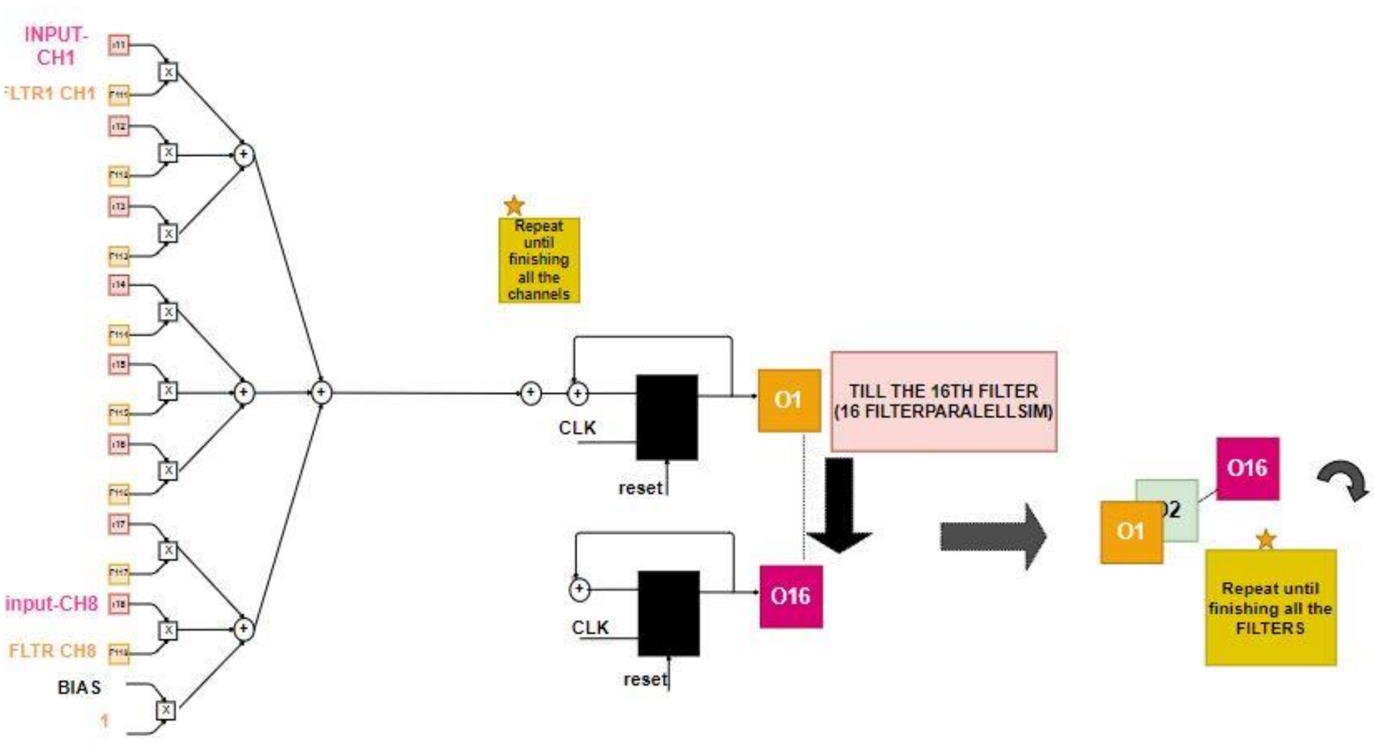

INPUT-CH1
FLTR1 CH1
input-CH8
FLTR CH8
BIAS
1
Repeat until finishing all the channels
CLK
reset
O1
TILL THE 16TH FILTER (16 FILTERPARALELLSIM)
O16
O2
Repeat until finishing all the FILTERS

FIGURE 59: CONVOLUTION OPERATION IN EXPAND_1X1 MODULE

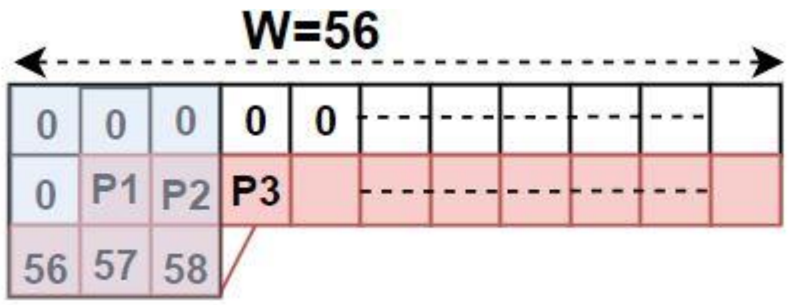


FIGURE 60: ZERO-PADDING IS DONE IN FIFO BUFFER BETWEEN SQUEEZE AND EXPAND_3X3

By using control signal this FIRE stage is adapted according to the running fire module due to variety number of filters and channels among different fire modules.

3.4 CONV_10 STAGE

CONV_10 layer includes 10 classes relative to application of internal security of a building from ImageNet data set.

3.4.1 CONVOLUTIONAL LAYER

In this layer there are 10 different filters with size 1x1, with stride equals 1 and number of channels equals 512, since the depth of the filters equals to the depth of the input feature map.

Parallelism

Since input feature map (IFM) of this layer has 512 layers, which is large number of channels, then we design Conv_10 layer to convolve 10 filters with 32 channels with same window of 32 channels of IFM in parallel in one clock cycle.

Input

The inputs of the convolution layer are

1. 10x32 pixels represent the 10 filters with size 1x1 across depth of 32 channels from weights memory.
2. 32 pixels represent the IFM parameters from the Ping-Pong memory.
3. Control signals.

Output

The output of the CONV_10 layer is 1x1x10 pixels of OFM after number of clock cycles =number of channels in CONV_10 module / 32 =512/32=16) clock cycles.

Methodology

In one cycle we convolve 32 channels of IFM with the opposite 32 channels of 10 filters, then accumulate the result of this convolution until we finish all the depth (number of channels) of the same window in OFM, then we can switch to the next window. As shown in figure 61, Conv_3x3 module is used to convolve 1x1x9 pixels of IPM with the opposite 1x1x9 pixels of one of the filters. Since we convolve 1x1x32 pixels of IPM with the opposite 1x1x32 pixels of 10 different filters at one cycle, then 4 instantiations of Conv_3x3 module are used and there results are accumulated with the next 32 channels till fishing all the channels of IFM , finishing one compound pixel in OFM, as shown in figure 62.

The 10 filters used represent the number of classes we want, as we choose 10 classes from 1000 classes related to the security of the buildings. In section# we will discuss in detail how we get the weights and the accuracy of the new pre trained model

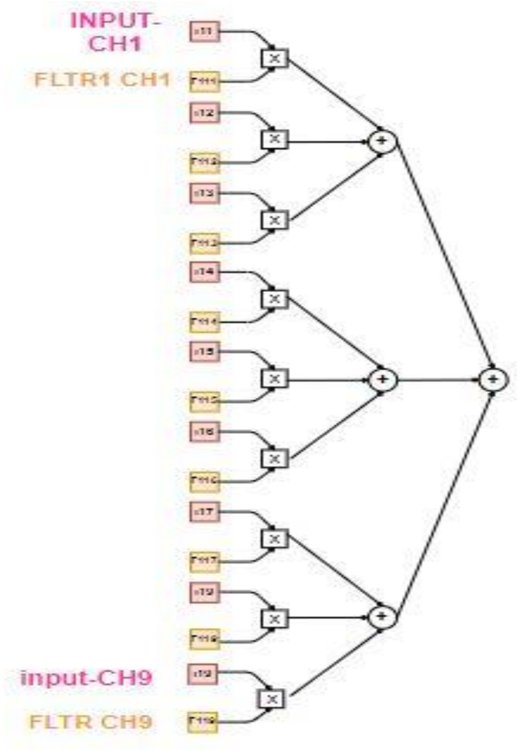


FIGURE 61: CONV_3X3 MODULE

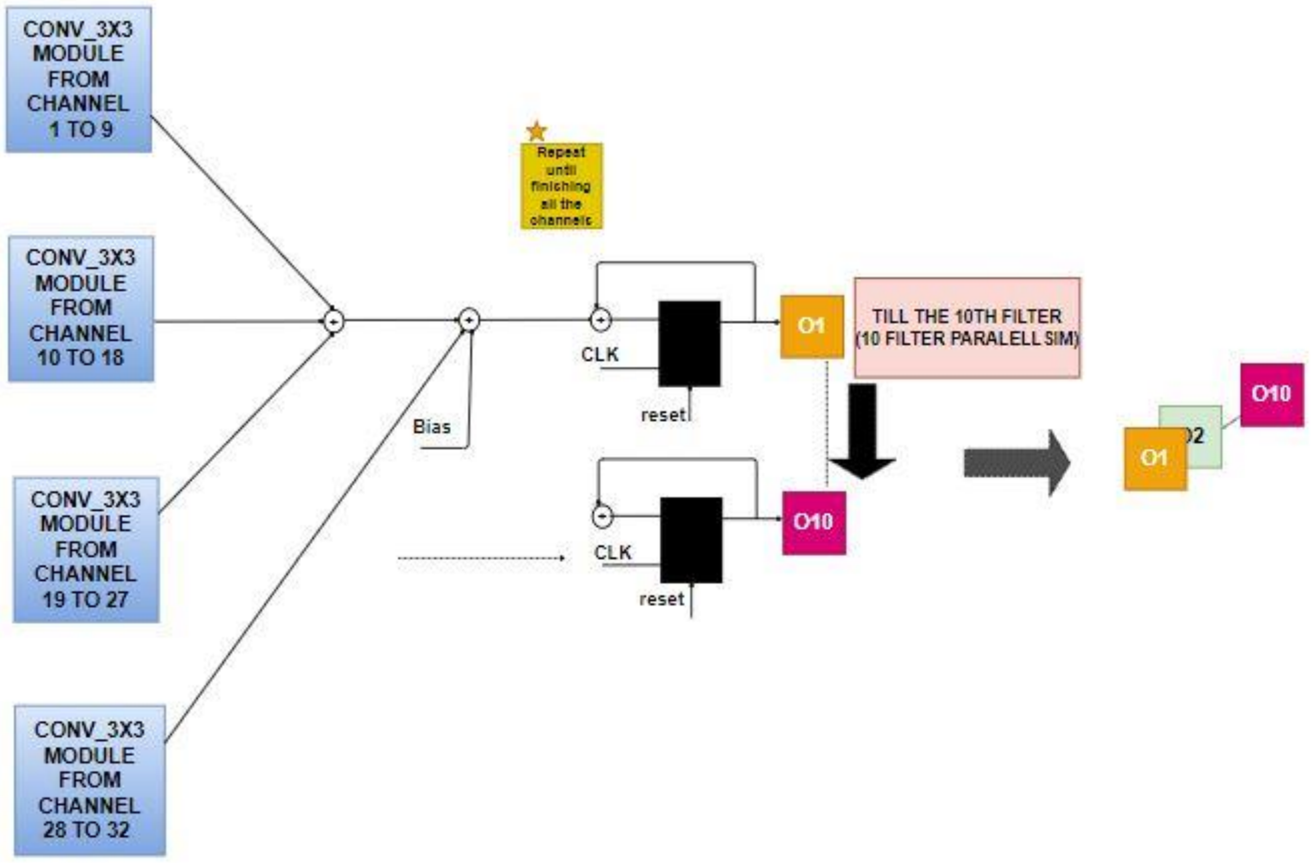

CONV_3X3 MODULE FROM CHANNEL 1 TO 9
CONV_3X3 MODULE FROM CHANNEL 10 TO 18
CONV_3X3 MODULE FROM CHANNEL 19 TO 27
CONV_3X3 MODULE FROM CHANNEL 28 TO 32
Repeat until finishing all the channels
Bias
CLK
reset
O1
TILL THE 10TH FILTER (10 FILTER PARALELL SIM)
CLK
reset
O10
O10
O2
O1

FIGURE 62: CONV_10 MODULE

3.4.2 GLOBAL AVERAGE-POOLING LAYER

In this final layer, it is required to get the probability of classification that represents how much the input image belongs to the 10 classes. This is done by getting the average of each channel $(\sum_{j=1}^{196} Pij)\big/_{196}$, where 'i' is the number of channels ranges from 1 to 10; and 'j' is the number of pixels in each channel which is equal to size of OFM of CONV_10 that equals 14x14.

This operation is done in our RTL design through 2 stages

1- Average pooling accumulator

2- Divider

1- AVERAGE POOLING ACCUMULATOR

INPUT

The inputs of the accumulator are

1. One pixel represents a parameter in a channel in OFM of CONV_10.
2. Control signals.

OUTPUT

The output is a pixel from accumulating the previous result of accumulation operation with new parameter from the OFM of CONV_10 for the same channel, as shown in figure 63.

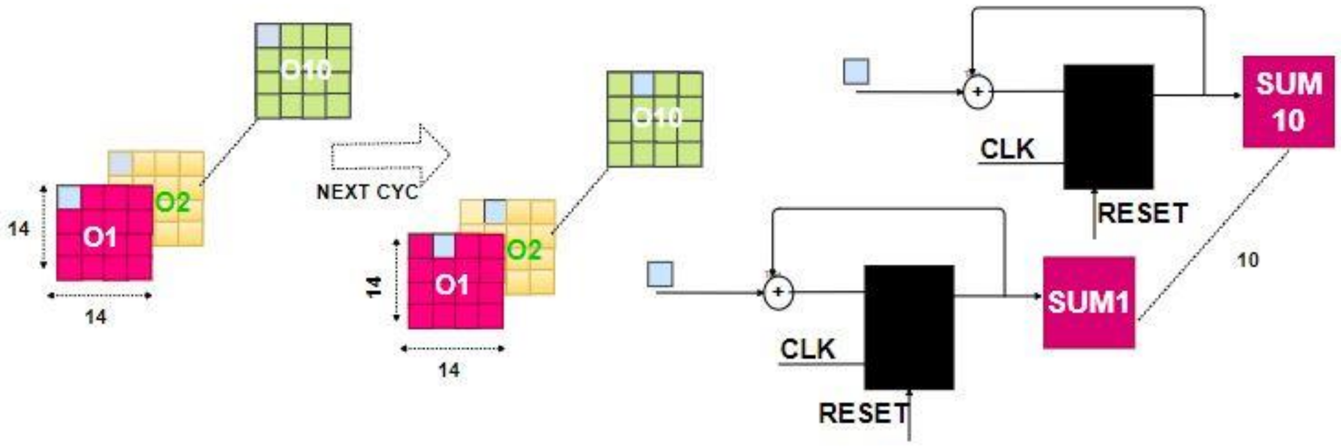


FIGURE 63: AVERAGE-POOLING ACCUMULATOR FOR 10 CHANNELS

2- DIVIDER

INPUT

The inputs of the divider are

1. Dividend
2. Divisor

3. Control signals

OUTPUT

A pixel represents the average of the parameters of each channel from OFM of CONV_10, which has size of 14x14 with 10 channels.

METHODOLOGY

Non-restoring division is used due to its modularity and simplicity rather than restoring division. As shown in figure 64, the flow chart of non-restoring division is done as a finite state machine (FSM).

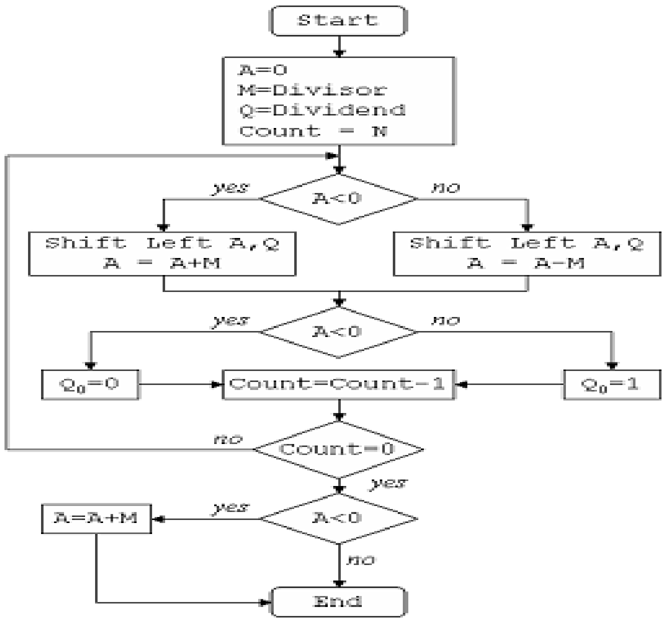


FIGURE 64: FLOW CHART OF NON-RESTORING DIVISION

3.5 HIERARCHY OF OUR DESIGN

Our Design as discussed in the previous section is divided into number of modules; each of them has its own control unit that ensures higher accuracy, modularity, easiness of the debugging and testing of each module as well as the whole design.

3.5.1 TOP MODULE

The only inputs to this module the global reset and the global clock signals from FPGA .As shown in figure 65, top module includes:

- Conv1_FIFO module includes Image_Mem module and FIFO_Conv1 module.

  - Image_Mem module reads the parameters of image and save it in FIFO_Conv1.

- Conv1 module includes Conv1_bias_Mem module and Conv1_Filters_Mem module.
  - The Conv1_Filters_Mem and Conv1_bias_Mem modules read the weights and biases saved in BRAMs to be used in convolution operation in Conv1 module that convolve a 3x3 winodw from FIFO_Conv1 with 32 filters of the 3 channels at a time.
- FIFO_Pooling module that has saved the IFM before pooling layer (after Conv1, Fire_3, Fire_5). It includes the FIFOs modules of different width sizes to ensure utilization as we discussed in detail earlier in section 3.2.
- Pooling module for getting the maximum parameter in the input, which is a 3x3 window of parameters, by comparing them as we discussed in section 3.2.
- PING_PONG module for saving the results of some layers and reading them back for the next layers. This module used the dual port BRAMS in the FPGA.
- Fire module includes Squeeze module, FIFO_Fire module (between Squeeze and Expand modules), and Expand module.
  - Squeeze module includes Squeeze_bias_Mem module and Squeeze_Filters_Mem module that reads the weights and biases saved in BRAMs that is used in convolution operation. Squeeze module convolves 16 channels of 8 filters at a time as discussed in section3.3.
  - Expand module includes Expand_bias_Mem module and Expand_Filters_Mem module that reads the weights and biases saved in BRAMs that is used in convolution operation. Expand module convolves a 3x3 window of IFM with that of 8 channels of 16 filters at a time as discussed in section3.3.

- Conv_10 module includes Conv10_bias_Mem module, Conv10_Filters_Mem module and Divider module.
  - The Conv10_Filters_Mem and Conv10_bias_Mem modules read the weights and biases saved in BRAMs to be used in convolution operation in Conv10 module that convolve a 1x1x32 channels with the 10 filters ,that represent the number of classes, at a time and then accumulate the result till finishing the whole channel of the 10 channels OPM, then using divider module for global average pooling operation as discussed in section3.4.

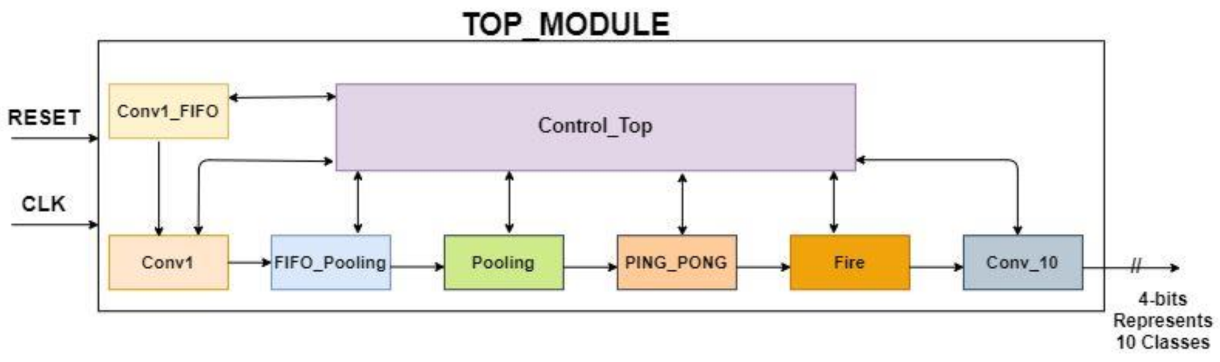


FIGURE 65: TOP_MODULE DESIGN

3.5.2 CONTROL TOP MODULE

The inputs to this module, as shown in figure 66:

- The global reset signals.
- The global clock signals.
- START_MASTER_CTRL signal: It is the output from ImageMem module to make Conv1 starts.
- New_pixel signal: It is the output from ImageMem module that indicates whether we could shift the window or not. If new_pixel equals 1 we will read a new window and convolve it with the first 32 filters, else when new_pixel equals 0 we will not and convolve with second 32 filters. (The same window convolves with 64 filters but we have in this stage 32 filters parallelism).
- Control_Top module includes Master_Control module which will described later in detail, and Delay_Regs to delay some control signals that is necessary for synchronization after multiply operation, adder tree and other delays due to pipelined design.

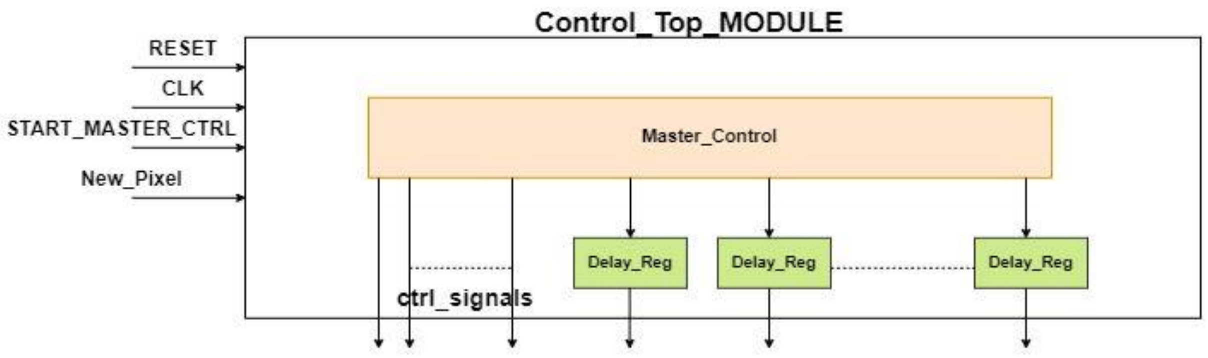


FIGURE 66: CONTROOL_TOP_MODULE DESIGN

3.5.3 MASTER CONTROL MODULE

The inputs to this module, as shown in figure 67:

- The global reset signals.
- The global clock signals.
- START_MASTER_CTRL signal
- New_pixel signal

As shown in figure 66, the Master_Control top is the main control units that includes all the sub-control units for other blocks and manage the order of each operation one after another. After ‘START_Master_control’ raised high, CONV_1 stage is started until the input image have finished completely (224x224 input pixel), by finishing CONV_1 stage following by POOLING stage and saving the OFM in Ping-Pong memory, ‘End_Conv_1’ signal is raised. Raising ‘END_CONV_1’ signal leads to raising ‘START_FIRE_2’ signal which enables the Fire stage doing its operation. Other fire modules will succeed one after another until FIRE_9 also by ‘CONF_START_FIRE’ signal and ’current_fire’ counter. If ’current_fire’ counter counts nine, then ‘START_CONV_10’ will raise leads to start CONV_10 stage following by global average pooling which gives the final result of classification of the input image.

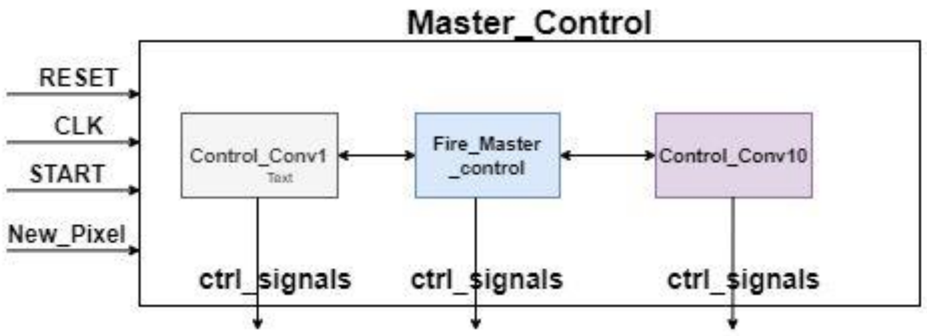


FIGURE 67: MASTER_CONTROL_MODULE DESIGN

3.5.3.1 CONV1 CONTROL

For CONV1 state there are 2 main operations as discussed in the previous section which are convolution operation and pooling operation. The CONV1_control module controls these operations and the intermediate stages through four main states which are:

- Pre-CONV1 state: that responsible for filling the FIFO buffer preceding CONV1 layer.
- CONV1 state
- CONV1 with pooling state: that takes place after filling the FIFO buffer preceding POOLING layer.
- POOLING state

3.5.3.2 FIRE MASTER CONTROL

The inputs to this module:

- The global reset signal.
- The global clock signal.
- CONF_START_FIRE

At this module there is an instantiation of Fire_control module that controls the operation of the fire module and the order of layers within the Fire stage. The Fire_Control module controls all the states in Fire module according whether this firemodule is followed by POOLING layer or not,

If it is followed by POOLING layer, there are five main states that are listed as:

- SQUEEZE state.
- SQUEEZE with EXPAND state: takes place after filling the FIFO buffer preceding EXPAND layer with the required zero-padding.
- SQUEEZE with EXPAND with POOLING state: takes place after filling the FIFO buffer preceding POOLING layer.
- Expand with POOLING state.
- POOLING state.

Else, there are 3 states only which are

- SQUEEZE state.
- SQUEEZE with EXPAND state.
- EXPAND state.

For different fire modules with different sizes, number of filters, number of channels and also the fires sometimes followed by different tracks, therefore there are 14 multiplexers as shown in figure 68:

1- Expand_Address_Bias_Depth, Squeeze_Address_Bias_Depth MUXs: To select the number of iterations to read the biases of SQUEEZE/EXPAND filters according to number of filters in each fire module.

2- Expand_Address_Filter_Depth, Squeeze_Address_Filter_Depth MUXs: To select the number of iterations to read the weights of SQUEEZE/EXPAND filters according to number of filters in each fire module.

3- Expand_Fire_Channels MUX: To select the number of iterations to finish all channels of IFM in each fire module.

4- Expand_Fire_Filters, Squeeze_Fire_Filters MUXs: To select the number of iterations to finish all Filters in each fire module.

5-Squeeze_Fire_Width_Squared, Expand_Fire_Width_Squared MUXs: To select the number of iterations to finish the whole pixels of IFM channel in each fire module.

6-Fire_Select MUX: To select how many sets from the 8 sets of FIFO_Conv1 will be selected according to the output of which filter.

7-Paralleism_Fire MUX: To order the output of Expand1x1 and Expand3x3 of the same fire in PINGPONG memory according to the number of filters in each fire.

8-Squeeze_Fire_Width MUX: To select the Width of IFM of Squeeze layer according to the current fire.

9-With_Pool: To select whether after the Expand layer in the current Fire, there is a Pooling operation or not.

10-Squeeze_Adress_Fire_Depth MUX: we select the number of iterations to finish all channels of IFM in each fire module.

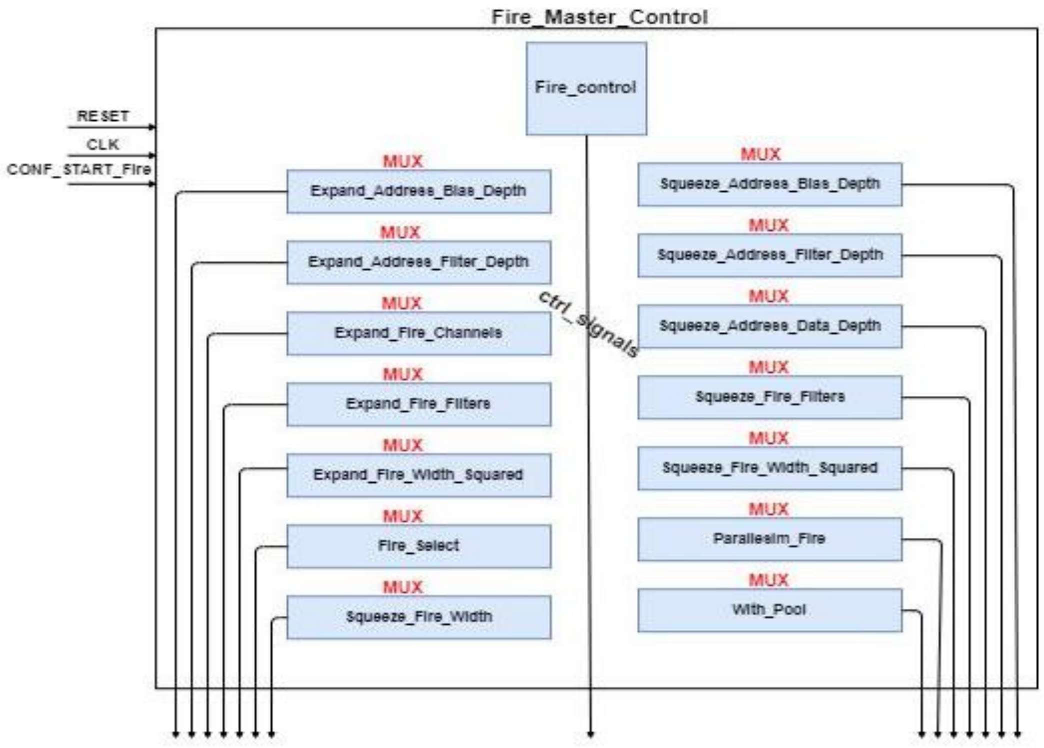

Fire_Master_Control
Fire_control
RESET
CLK
CONF_START_Fire
MUX
Expand_Address_Bias_Depth
Expand_Address_Filter_Depth
Expand_Fire_Channels
Expand_Fire_Filters
Expand_Fire_Width_Squared
Fire_Select
Squeeze_Fire_Width
Squeeze_Address_Bias_Depth
Squeeze_Address_Filter_Depth
Squeeze_Address_Data_Depth
Squeeze_Fire_Filters
Squeeze_Fire_Width_Squared
Parallesim_Fire
With_Pool
ctrl_signals

FIGURE 68: FIRE_MASTER_CONTROL MODULE DESIGN

3.6 CONV10 Control

For CONV10 state there is one main operation as discussed in the previous section which is convolution operation with 10 filters, accumulator and divider for global average pooling. The CONV10_control module controls this operation through only one main state which is:

- CONV10 state

3.7 SUMMARY

In this chapter, we have discussed in details our design for implementing SqueezeNet on FPGA through explaining the different stages of the architecture and their control. In chapter synthesis and optimizations that are done to this design will be presented.

CHAPTER 4. SYNTHESIS AND OPTIMIZATIONS

In this chapter, the goal is to enhance the design in terms of timing and area along with fixing the synthesis issues and solve any simulation and synthesis mismatch.

This phase started with a utilization of 88% LUTs and a clock speed of 100 MHz with zero slack. It ended with a utilization of 22% LUTs and a clock speed of 200 MHz with 1.4ns positive slack. In section 2.3 the major enhancements in timing and area will be discussed in detail and with a quick overview of minor ones. In section 4, the final results of the synthesis phase will be discussed as well.

4.1 TIMING ENHANCEMENTS

4.1.1 ENHANCE BRAM TIMING

To follow the standard language template of BRAM inference from the FPGA vendor user guide which recommends including output registers to every BRAM to enhance, and separate the logic delay of the BRAM from other combinational delays after it. Adding this output register allow the BRAM instance to work at a frequency up to 400 MHz

As shown in figure 69, the internal structure of a BRAM module and the output registers.

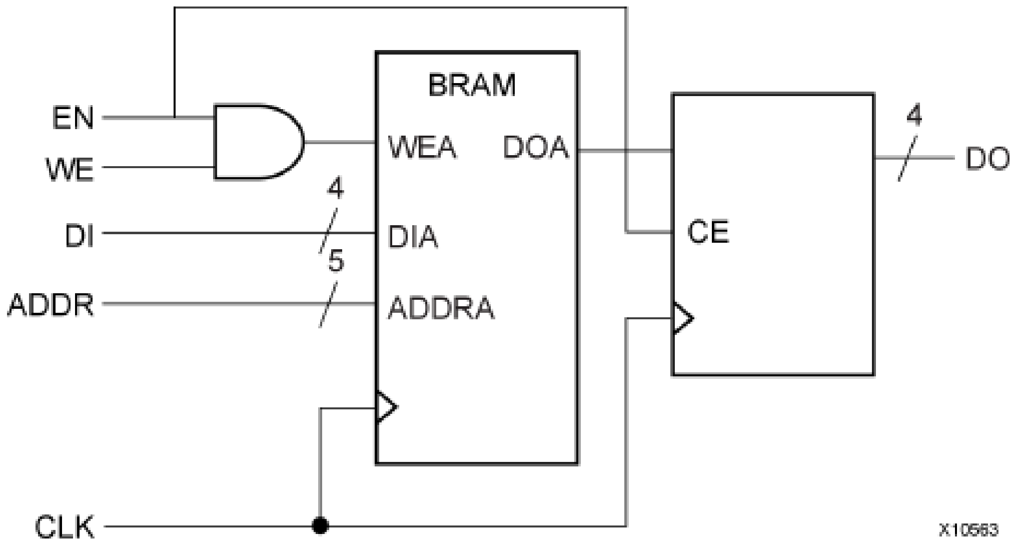


FIGURE 69: STRUCTURE OF BRAM MODULE

4.1.2 Enhance DSP timing and power

To follow the standard language template of DSP inference from the FPGA vendor user guide. A fully pipelined DSP is used to separates every stage of the DSP using registers, as shown in figure 70. An input register is added as well to enhance the power of the module which is also suggested by the FPGA vendor user guide.

DSPs can perform this equation

$$P = (A + D) \times B + C$$

In our design, DSPs are used either as a multiplier $(A \times B)$ or as a multiply and add$(A \times B + C)$.

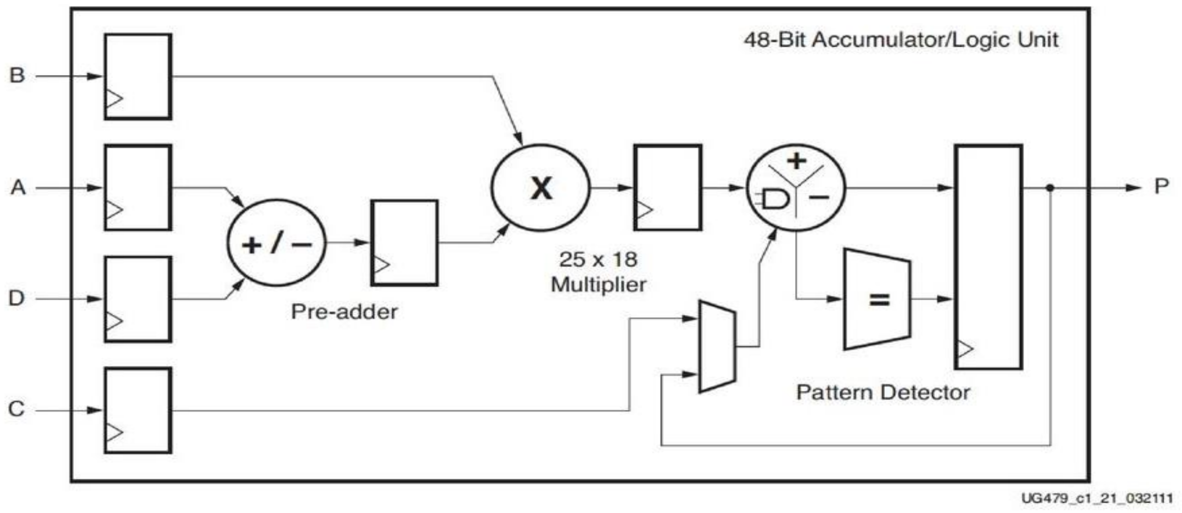


FIGURE 70: STRUCTURE OF DSP MODULE

4.1.3 Breaking and Pipelining large combinational paths

Long chains of logic elements (LUTs) cause a large timing path. Some of these chains contained redundant logic which was removed enhancing both timing and area. Other chains were fixed by separating each combinational block from the other using registers. Other chains were fixed by pipelining the logic such as the CSA adder which was converted to a multi-cycle adder instead of single cycle to enhance timing while sacrificing some area for the added registers, as shown in figure 71.

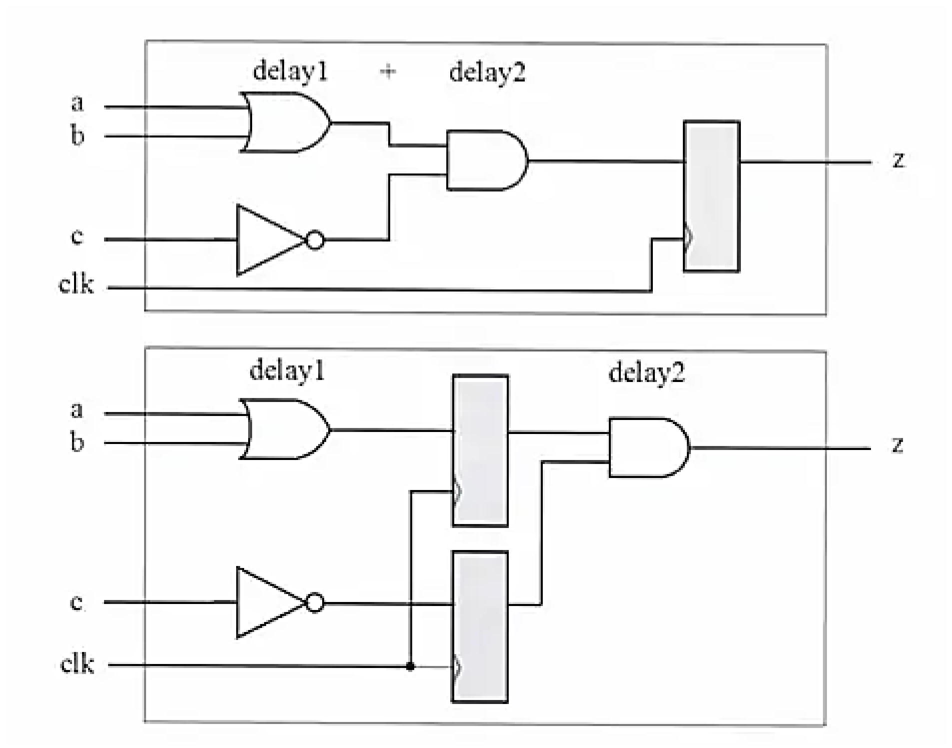


FIGURE 71: NON-PIPELINED/PIPELINED LOGIC UNITS

4.1.4 USING ONE HOT ENCODING FOR FSM

One hot FSM encoding uses a separate register for each state. The state register is connected directly to the FSM outputs providing the fastest clock to out timing and are simple, because, with other encoding, logic gates are used to “decode”, for example, an 8 state with 3-bit representation into one of the eight states, whereas with one-hot encoding there is nothing to decode. The state is connected directly to the one bit corresponding to the state which is the out of the register.

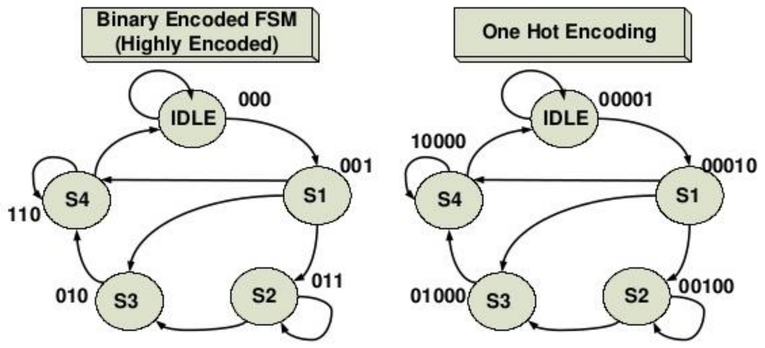


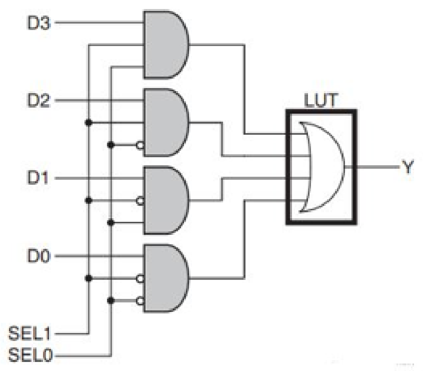
FIGURE 72: BINARY/HOT ENCODING

4.2 Area enhancement

4.2.1 Using MUX primitive instead of LUTs

The FPGA fabric contains Mux primitive along with LUTs, as shown in figure 73. In order to direct the FPGA synthesizer to use Mux primitive instead of creating mux using LUTs, the full case needs to be provided to the mux otherwise the synthesizer will optimize the code and use LUTs instead.

The problem with Mux primitive is that they cause congestion during placement and routing. This will be discussed in chapter 7.



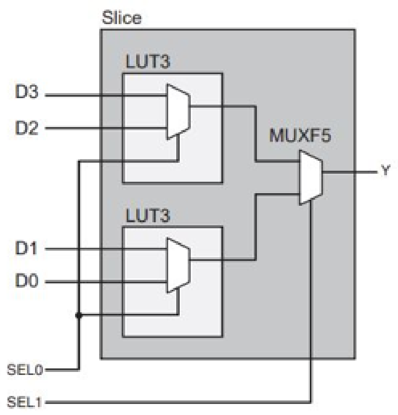


FIGURE 73: MUX PRIMITIVE ALONG WITH LUTS

4.2.2 Using don’t care in default blocks

Don’t care in Verilog can be used to tell the synthesizer that this logic is not used or unreachable and can be optimized away. Therefore, don’t cares are assigned to all variables in default block in case statement.

4.3 Final results

4.3.1 Area summary

The final utilization for the whole design and each module is shown in Table.1. The biggest blocks are the pooling FIFO and the Fire blocks. Fire blocks have most of the BRAMs and DSPs due to the large number of weights stored in ROMs and the large number of multipliers.

Table 3: Utilization of LUTs, Registers and MUXs for our design

	LUTs			Registers			Mux		
	Count	% of FPGA	% of Design	Count	% of FPGA	% of Design	Count	% of FPGA	% of Design
Image ROM	920	0.21%	0.94%	450	0.05%	0.28%	0	0%	0%
Conv 1	13793	3.18%	14.2%	19644	2.27%	12.3%	0	0%	0%
Fire	32677	7.54%	33.7%	44452	5.13%	27.8%	1080	0.5%	19%
Conv 10	7708	1.78%	7.95%	10065	2.27%	6.3%	0	0%	0%
FIFO Pooling	36480	8.42%	37.6%	82084	9.47%	51.4%	4320	1.99%	76.1%
Pooling	1248	0.29%	1.29%	1920	0.22%	1.2%	0	0%	0%
Ping-Pong	512	0.12%	0.53%	0	0%	0%	0	0%	0%
Control unit	3844	0.89%	3.96%	1153	0.13%	0.72%	274	0.13%	4.83%
Total	96990	22.4%	100%	159772	18.4%	100%	5674	2.62%	100%

Table 4: Utilization of BRAMs and DSPs for our design

	BRAMs			DSP		
	Count	% of FPGA	% of Design	Count	% of FPGA	% of Design
Image ROM	96	34.83%	%9.7	0	0%	0%
Conv 1	0	0%	0%	864	24%	32.5%
Fire	384	26.12%	38.7%	1464	40%	55.1%
Conv 10	0	0%	0%	330	9.1%	12.4%
Ping-Pong	512	38.83%	51.6%	0	0%	0%
Total	992	67.48%	100%	2658	73.8%	100%

Table 5: Utilization of LUTs and REGs for carry save adder and divider

	Count	LUTs per module	Total LUTs (% of Design)	Regs Per module	Total Regs %
CSA	1308	32	41856(43.2%)	0	0(0%)
Divider	10	141	1410(1.45%)	71	710(0.44%)

4.3.2 TIMING SUMMARY

The critical paths for each combinational module. The control unit FSM is the most complex having the longest chain of logic which is 9 in Conv 1 control and 8 in Fire control. The CSA adder and the 3-input comparator follow with a logic level of 7 for both.

4.4 SUMMARY

In this chapter, an overview over synthesis results of area and timing is discussed with different methods that are used to improve timing, area, and power. The next chapter focuses on the output of the implementation step.

CHAPTER 5. IMPLEMENTATION

In this chapter different implementations with different clock periods will be discussed in the following sections.

5.1 CLOCK PERIOD = 10 NS:

The design was implemented at a frequency of 100 MHz and the tool strategy was “spread-logic = high “to avoid creating any congestion regions due to the design has low utilization and it was meeting the timing constraints but there was a high wire delay which was the motivation to improve the frequency to 200 MHZ and get benefit from this high wire delay

5.2 CLOCK PERIOD = 6 NS:

5.2.1 REASONS FOR HIGH WIRE DELAY

1. Cross-boundary optimization of synthesis
2. High fan-out
3. LUT combining
4. MUXF7, MUXF8
5. Bad Floor-planning
6. Wrong directive used in placement

5.2.2 TIMING CLOSURE TECHNIQUES

CREATE CONSTRAINT FILE:

SYNTHESIS CONSTRAINT FILE:

- Create clock
- Define clock interactions
- Set input and output delays
- Set IOB Buffer registers to improve timing

PHYSICAL CONSTRAINT FILE:

- Define input differential clock pins
- Define input and output pins connected to FPGA
- Floor planning Style

5.2.3 ANALYZING SYNTHESIS RESULT

- Create slack histogram to analyze the modules which don't meet the timing analysis before placement as a first intuition
- Analyze complexity and congestion that will be seen by the placer and the router to have an image to re-synth with another technique or stop crossboundary optimization at certain module or re-design any certain module

5.2.4 PLACEMENT

- Try most of the placement directives techniques and didn't expect that it will solve all the problems but this will give an initiation about the best directive consistent with your design which has the lowest worst negative slack

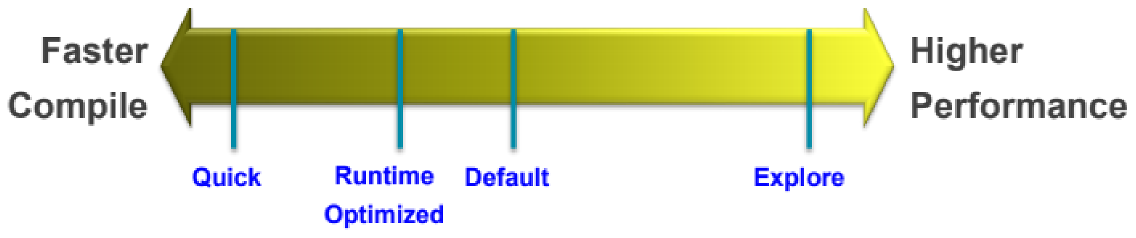


FIGURE74: PLACEMENT DIRECTIVES

5.2.5 CONGESTION:

Congestion has two reasons high pin density, and high utilization of routing resources

- Placer Congestion: can be reduced with tool suggestions commands to know the reason of the congestion in the design and replacement after solving it and see the result again and therefore on.
- Router Congestion: routing detours are used to handle congestion at the expense of timing.

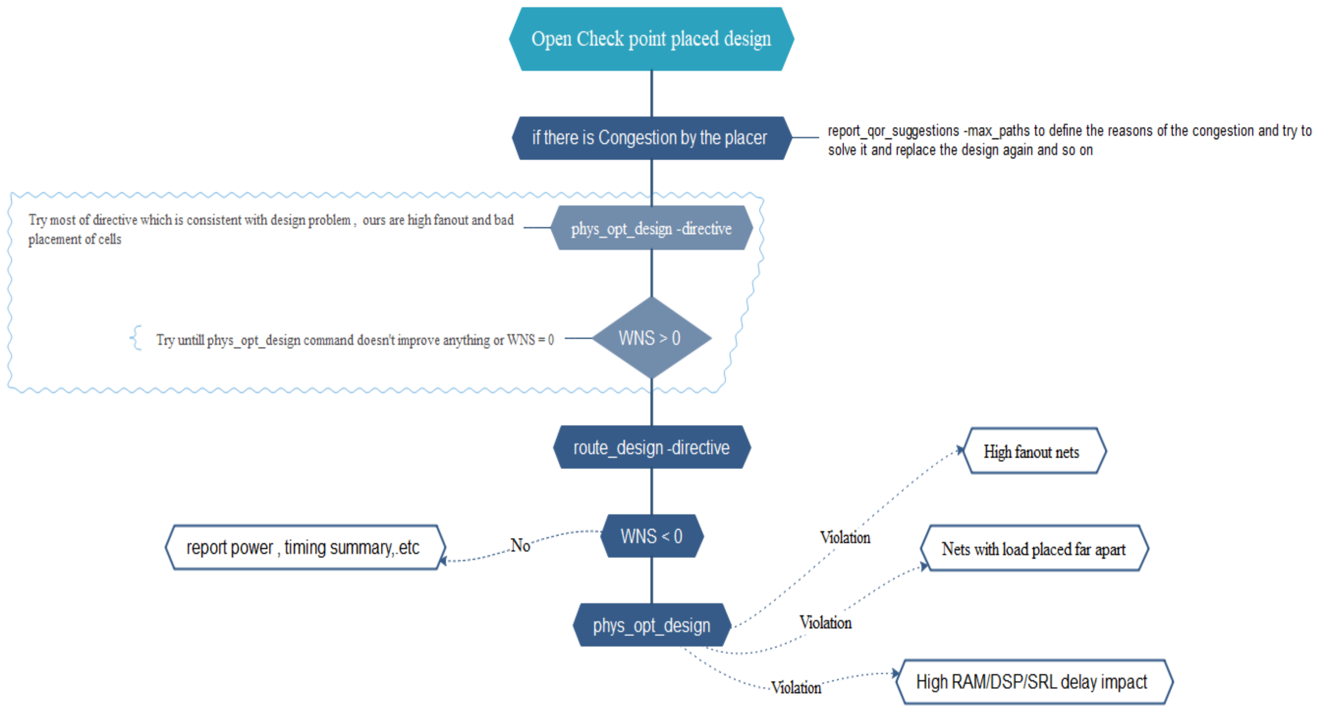

Open Check point placed design
if there is Congestion by the placer
report_qor_suggestions -max_paths to define the reasons of the congestion and try to solve it and replace the design again and so on
Try most of directive which is consistent with design problem , ours are high fanout and bad placement of cells
phys_opt_design -directive
Try untill phys_opt_design command doesn't improve anything or WNS = 0
WNS > 0
route_design -directive
High fanout nets
report power , timing summary,.etc
No
WNS < 0
Violation
Nets with load placed far apart
phys_opt_design
Violation
Violation
High RAM/DSP/SRL delay impact

FIGURE 75: TIMING ANALYSIS USING VIVADO TCL COMMANDS

5.3 FPGA FLOOR-PLANNING:

If the high wire delay problem isn't solved then it can lead to use the FPGA floor-planning to place modules that are communicating side by side to reduce the wire length and that can be achieved using PBLOCK option in VIVADO GUI to define a certain area for modules that are communicating to each other close to each other's and repeat the above flowchart sequence again until the wire delay is reduced.

as shown in the following figure : Conv1 and Fire modules is placed side by side and Data mem and MASTER_CTRL_UNIT placed as children for Fire PBLOCK in the middle of the design as they are sharing resources between all modules and with inferring the tool for timing optimization it will set the FIFO_POOLING and the POOLING module between Conv1 and the Fire PBLOCK to reduce the wire delay and place conv10 close to the MASTER_CTRL_UNIT to reduce the delay of the needed signals from the MASTER_CTRL_UNIT.

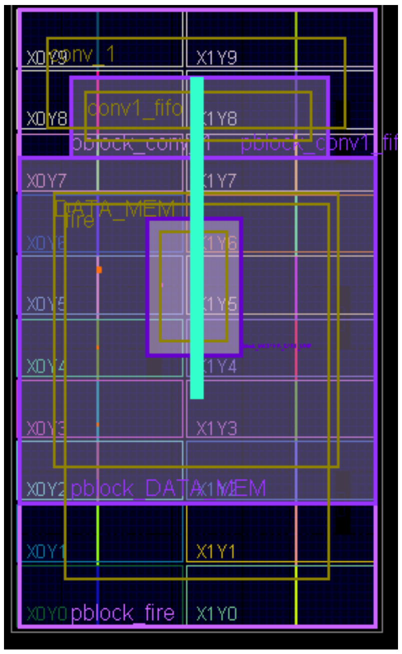
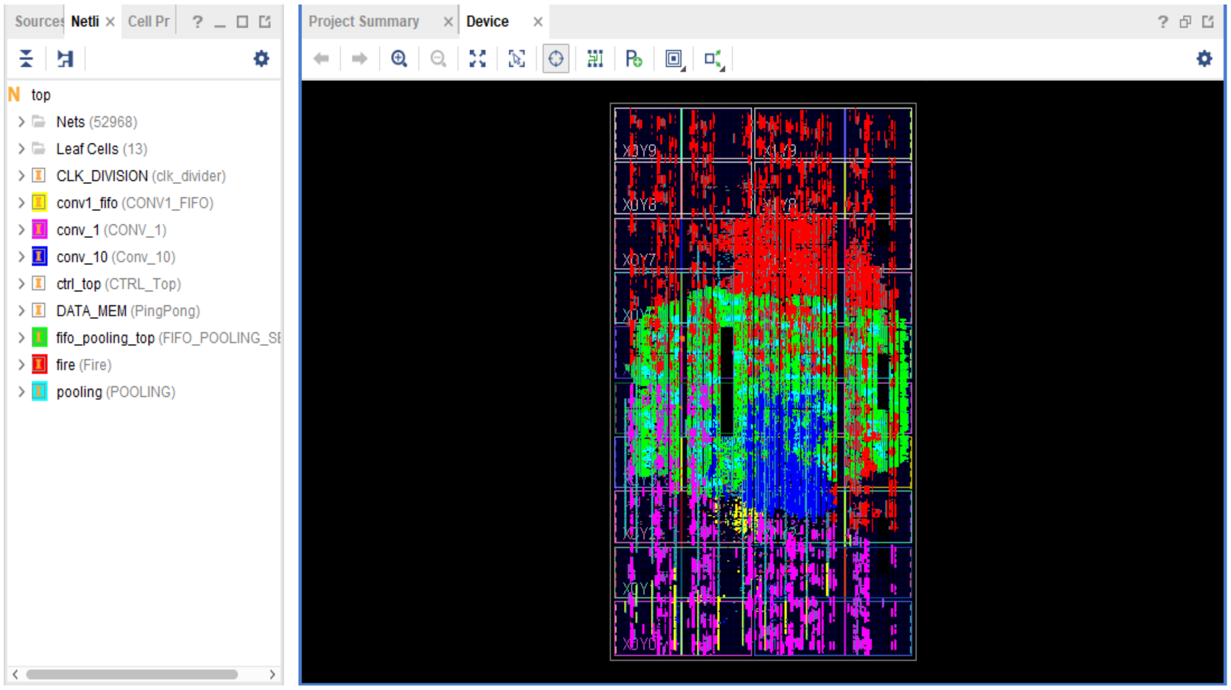


FIGURE 76: FLOOR PLANNING FOR DESIGN



N top
- Nets (52968)
- Leaf Cells (13)
- CLK_DIVISION (clk_divider)
- conv1_fifo (CONV1_FIFO)
- conv_1 (CONV_1)
- conv_10 (Conv_10)
- ctrl_top (CTRL_Top)
- DATA_MEM (PingPong)
- fifo_pooling_top (FIFO_POOLING_SE
- fire (Fire)
- pooling (POOLING)

FIGURE 77: PLACEMENT BEFORE FLOOR PLANNING

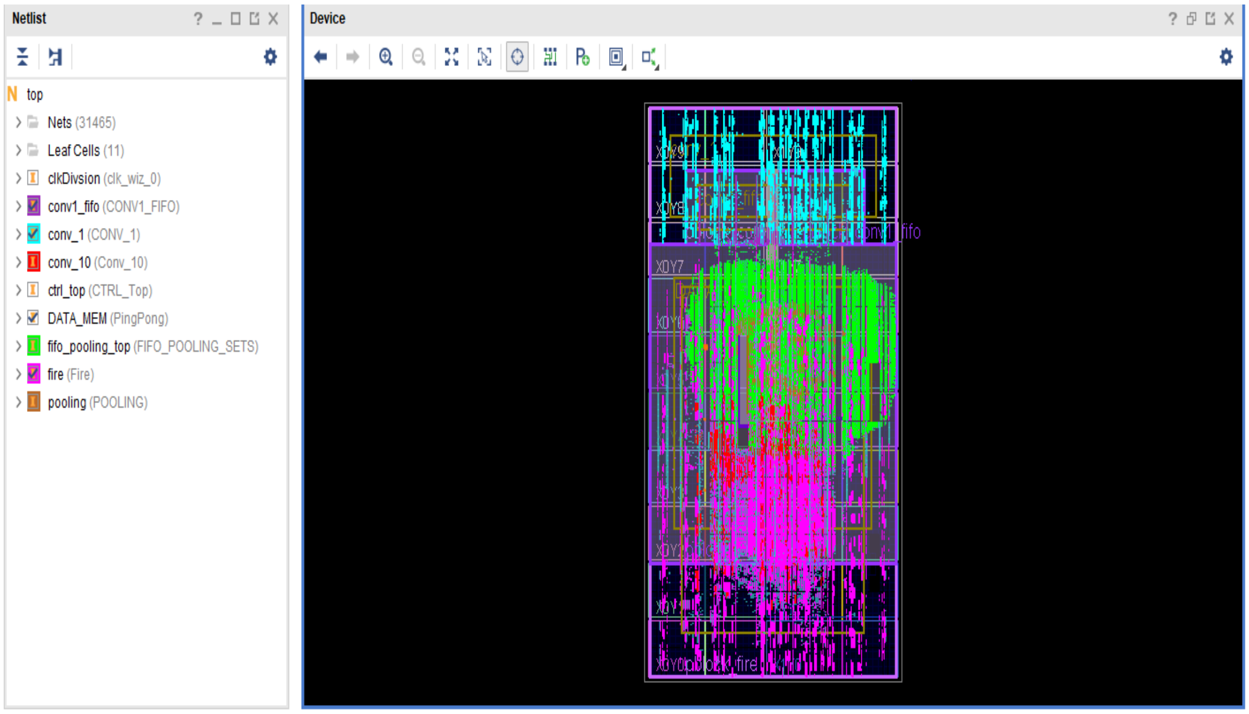


FIGURE 78: PLACEMENT AFTER FLOOR PLANNING

5.4 SUMMARY

This chapter discussed the output of implementation phase, the struggles and problems in design and different tips to complete successfully the floor planning and routing. In next chapter software results and optimization technique to achieve the highest possible accuracy are discussed.

CHAPTER 6. SOFTWARE OPTIMIZATIONS

In this chapter, the software modifications and optimizations will be presented in the following sections to efficiently implement Squeeze-Net on FPGA.

6.1 TRANSFER LEARNING

Transfer learning consists of taking features learned on one problem, and leveraging them on a new, similar problem. It is usually done for tasks where your dataset has too little data to train a full-scale model from scratch.

The most common steps of transfer learning in the context of deep learning are the following workflow:

1. Take layers from a previously trained model.
2. Freeze them, to avoid destroying any of the information they contain during future training rounds.
3. Add some new, trainable layers on top of the frozen layers. They will learn to turn the old features into predictions on a new dataset.
4. Train the new layers on your dataset.

The TensorFlow option seemed to be the most promising since it could train a Convolutional Neural Network very quickly using the GPU and leverage the extremely large data set provided by the ImageNet Challenge database. The goal of developing a model in TensorFlow was achieved to load the re-trained weights in the design memory.

A small data set of 10 classes from ImageNet dataset was used to train Squeeze-Net Convolutional Neural Network. These classes were selected for security of buildings applications as luggage scanner or detecting some edged, bladed weapons as listed in table 6 and figure 78.

Table 6: 10 Classes Used to Train the Squeeze-Net Model

Class Number	ImageNet ID	Class Description
1	n02749479	assault rifle
2	n02951585	can opener
3	n03000684	chainsaw

4	n03041632	cleaver
5	n03109150	corkscrew
6	n03481172	hammer
7	n03498962	hatchet
8	n03658185	letter opener
9	n04090263	rifle
10	n04154565	screwdriver

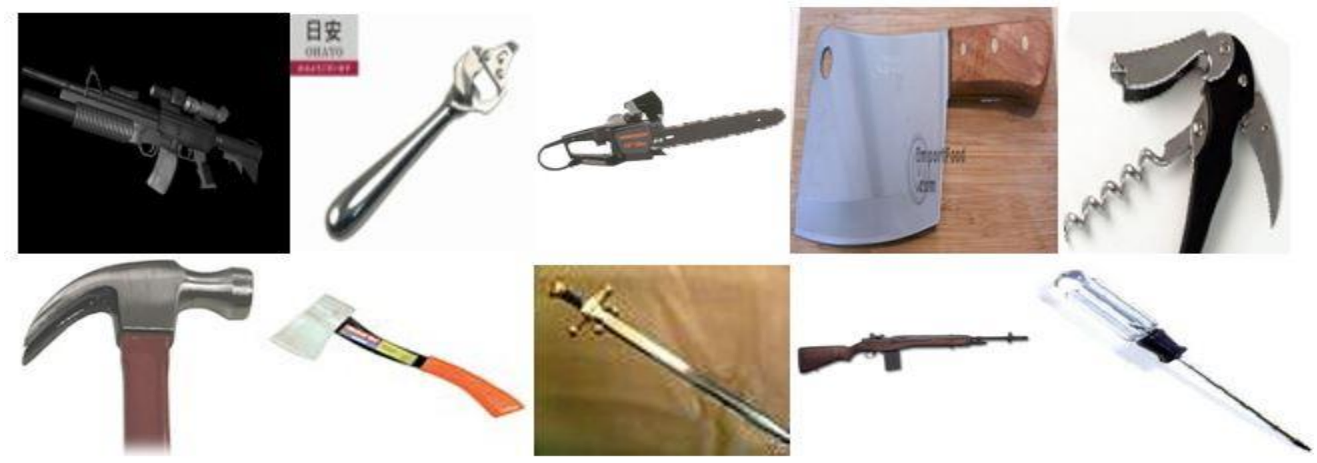
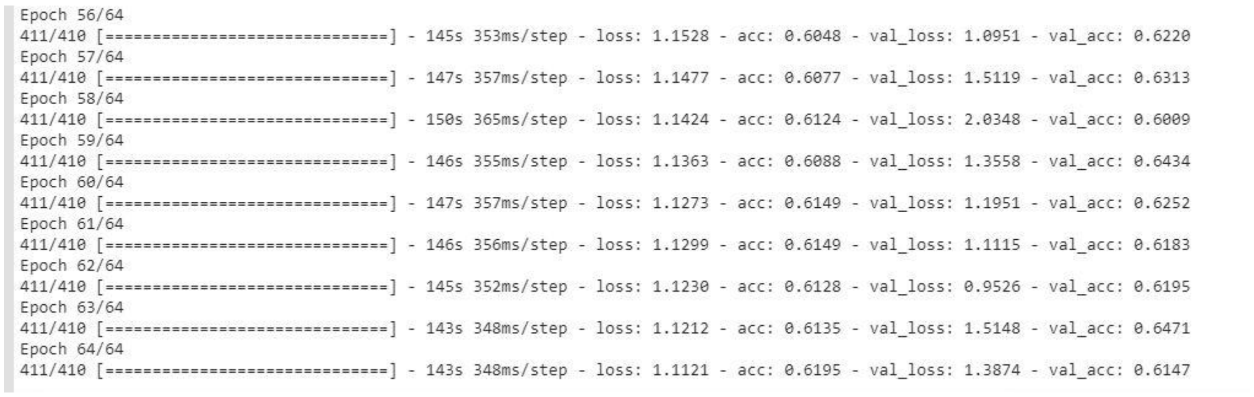


FIGURE 78: 10 RANDOMLY SELECTED IMAGES FROM 10 CLASSES

Keras is an open-source neural-network library written in Python and it is capable of running on top of TensorFlow, to run this code, these codes can be seen at the links in the appendix. Firstly, the trained weights of the pre-trained model from CONV1 layer to Fire 9 layer have been included. Secondly, freezing all the layers expect Fire 9 layer for retraining it. Then, Adding CONV10 layer with 10 filters, represents 10 classes instead of 1000, of size 1x1x512. Using 1000 images for each class from ImageNet data set, 80% as the training set for the Squeeze-Net CNN (9857 images for training set) and 20% as validation set (2468 images for validation set), the network was trained over 4 iterations. During these iterations, the parameters of the model have varied to improve the validation accuracy. The final parameters that results in validation accuracy 61.47% as shown in figure 79 are listed as below:

- Regularization = 0.0 (No Regularization)
- Learning Rate = 0.0001

- Weight Scale = 1/255
- Batch Size = 32
- Epochs = 64
- Optimizer = Adams optimizer



```
Epoch 56/64
411/410 [==============================] - 145s 353ms/step - loss: 1.1528 - acc: 0.6048 - val_loss: 1.0951 - val_acc: 0.6220
Epoch 57/64
411/410 [==============================] - 147s 357ms/step - loss: 1.1477 - acc: 0.6077 - val_loss: 1.5119 - val_acc: 0.6313
Epoch 58/64
411/410 [==============================] - 150s 365ms/step - loss: 1.1424 - acc: 0.6124 - val_loss: 2.0348 - val_acc: 0.6009
Epoch 59/64
411/410 [==============================] - 146s 355ms/step - loss: 1.1363 - acc: 0.6088 - val_loss: 1.3558 - val_acc: 0.6434
Epoch 60/64
411/410 [==============================] - 147s 357ms/step - loss: 1.1273 - acc: 0.6149 - val_loss: 1.1951 - val_acc: 0.6252
Epoch 61/64
411/410 [==============================] - 146s 356ms/step - loss: 1.1299 - acc: 0.6149 - val_loss: 1.1115 - val_acc: 0.6183
Epoch 62/64
411/410 [==============================] - 145s 352ms/step - loss: 1.1230 - acc: 0.6128 - val_loss: 0.9526 - val_acc: 0.6195
Epoch 63/64
411/410 [==============================] - 143s 348ms/step - loss: 1.1212 - acc: 0.6135 - val_loss: 1.5148 - val_acc: 0.6471
Epoch 64/64
411/410 [==============================] - 143s 348ms/step - loss: 1.1121 - acc: 0.6195 - val_loss: 1.3874 - val_acc: 0.6147
```

FIGURE 79: ACCURACY AFTER 4 ITERATIONS EACH OF 64 EPOCHS

6.2 OVERFLOW

This design uses fixed-point representation. Some numbers overflowed causing numbers to change in sign which caused a significant loss in accuracy. We tried several solutions to this issue. In the following section we will discuss them and show the advantages and disadvantages of each.

6.2.1 SATURATION INSTEAD OF WRAPPING

In order to stop numbers from changing sign, a maximum limit on numbers was placed. For example, if the fixed-point representation has 4 bits for integer part then the maximum will be$2^{4-1} - 1 = 7$, and any number that exceeds this limit will be brought back to the maximum. This led to some loss in accuracy

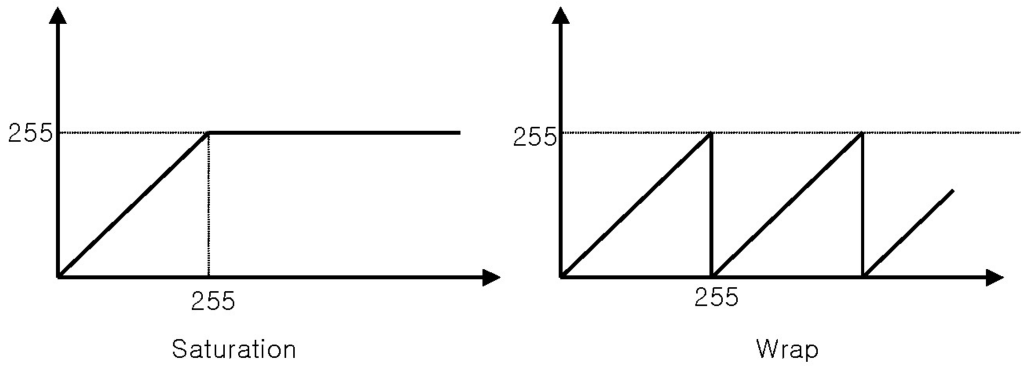


FIGURE 80: SATURATION VS WRAPPING

6.2.2 Clipped RELU

Applying a maximum limit on hardware causes some loss in accuracy that's why it was better to place the limit in software and retrain the network to deal with these new limits.

A different version of RELU activation function was utilized, it is called Clipped RELU as shown in Fig. 81.

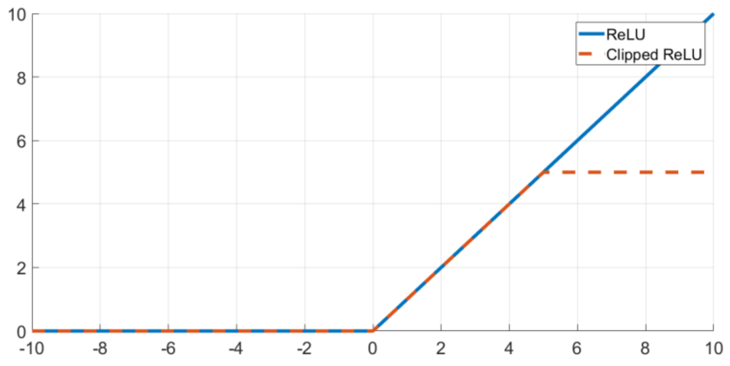


FIGURE 28: CLIPPED RELU VS NORMAL RELU

After experimenting on different limits, it was shown that a limit of 8 or less will cause a loss of 4% in accuracy, and the more we increase the limit the higher the accuracy. Fig.82 shows the validation accuracy versus iteration for different limits.

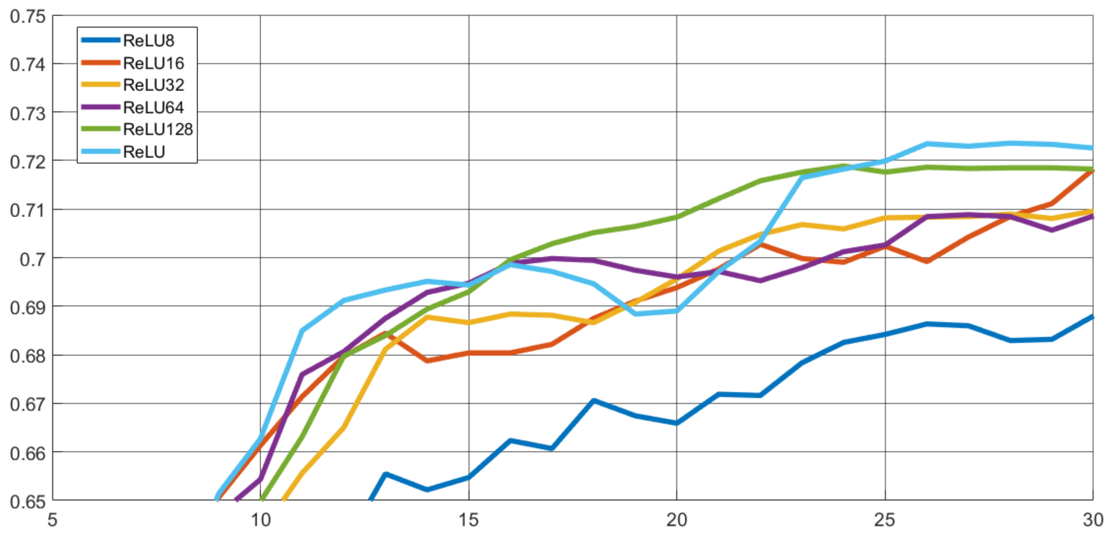


FIGURE 29: VALDIATION ACCURACY FOR DIFFERENT LIMITS ON RELU FUNCTION. MOVING AVERAGE WAS APPLIED TO DATA.

After calculating the accuracy in hardware, it was found that the accuracy is still lower compared to the software accuracy.

6.2.3 Clipped RELU and Dynamic fixed point

The best solution was to combine the previous solution with a dynamic fixed point representation.

First layers in the network have lower range of numbers around -10, and 10 while last layers have higher range. Different representation was used for different layers, i.e. giving the first layer fewer bits for integer and the last layers more bits. The First layers were trained on small limits and the limits were increased as for the last layers. Table 7 shows the limits for every layer.

Table 7: Maxiumum limit for each layer

Layer	Conv1, Fire2, Fire3	Fire4, Fire5	Fire6, Fire7	Fire8, Fire9, Conv10
Maximum	4	16	64	128

This solution achieved the closest hardware accuracy compared to the software. And the loss in software accuracy was almost negligible.

6.3 SUMMARY

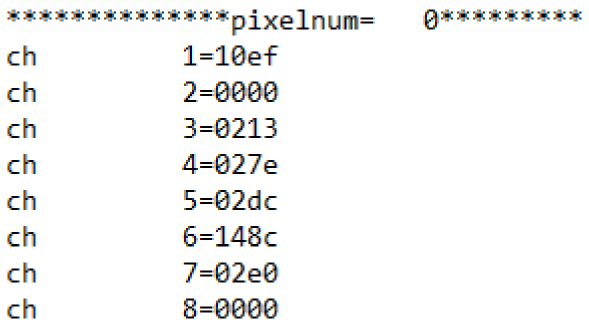
This chapter discussed the purposed application on this design and steps to perform transfer learning successfully and clipped RELU methods which used to achieve acceptable accuracy.in next chapter the final results from timing, hardware simulation and power is overviewed.

CHAPTER 7. RESULTS

In this chapter results of simulation and comparison with other works will be presented.

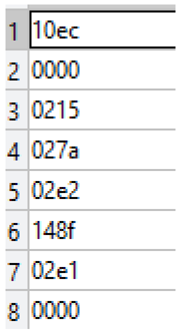
7.1 SIMULATION RESULTS

To test the simulation results a complete MATLAB code is written for squeeze-net architecture to compare the intermediate values between different layers such as squeeze and expand or between different fires modules and to predict the degradation in accuracy due to limited 16-bits fixed number representation.



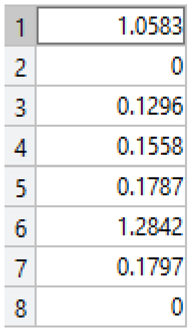
```
**************pixelnum=   0*********
ch          1=10ef
ch          2=0000
ch          3=0213
ch          4=027e
ch          5=02dc
ch          6=148c
ch          7=02e0
ch          8=0000
```

FIGURE 83-A: VALUES OF 10 CLASSES FROM RTL MODEL IN HEX



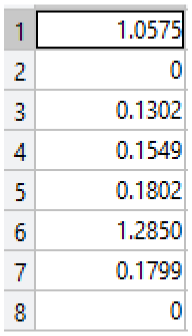
1	10ec
2	0000
3	0215
4	027a
5	02e2
6	148f
7	02e1
8	0000

FIGURE 83-B: VALUES OF 10 CLASSES FROM MATLAB MODEL



1	1.0583
2	0
3	0.1296
4	0.1558
5	0.1787
6	1.2842
7	0.1797
8	0

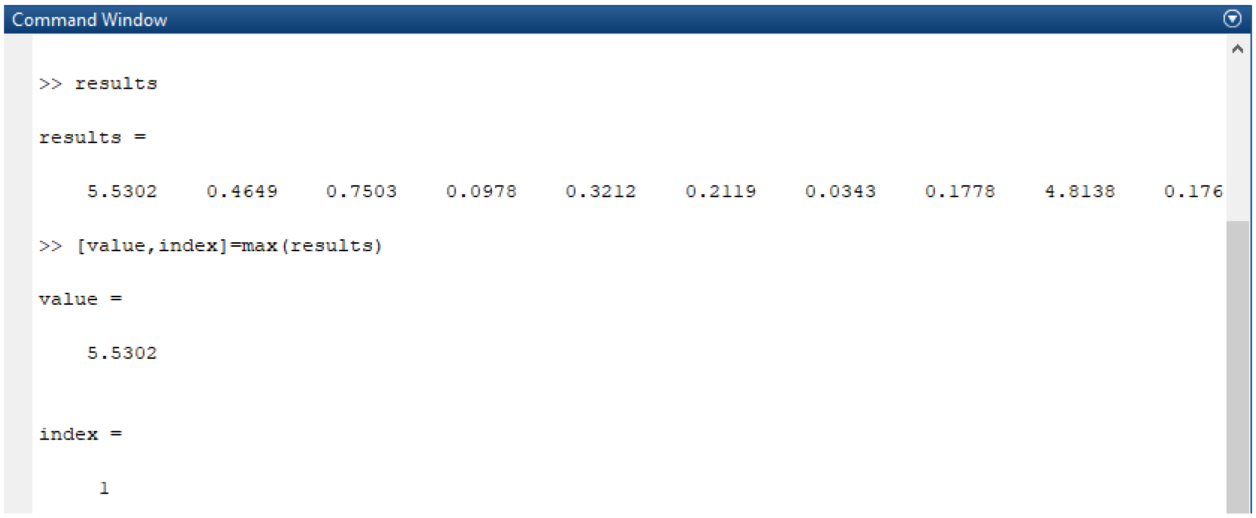
FIGURE 83-C: VALUES OF 10 CLASSES FROM RTL MODEL IN DECIMAL



1	1.0575
2	0
3	0.1302
4	0.1549
5	0.1802
6	1.2850
7	0.1799
8	0

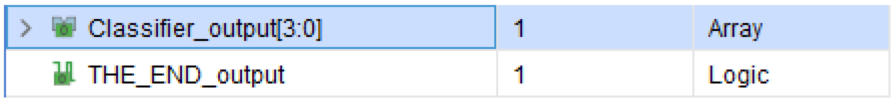
FIGURE 83-D: VALUES FROM MATLAB MODEL

As shown in figures 81, the intermediate values from hardware model and MATLAB code is very close and the difference can be neglected and approximated in both bases' hexadecimal and decimal. The purpose of CNN is classifying the image therefore to determine the performance of hardware design it should choose the correct class. As shown in figures 82, the hardware design successfully determined the class.



```
>> results

results =

    5.5302    0.4649    0.7503    0.0978    0.3212    0.2119    0.0343    0.1778    4.8138    0.176

>> [value,index]=max(results)

value =

    5.5302


index =

     1
```

FIGURE 84-A: CLASS IDENTIFICATION FROM MATLAB MODEL



Classifier_output[3:0]	1	Array
THE_END_output	1	Logic

FIGURE 84-B: CLASS IDENTIFICATION FROM RTL MODEL

7.2 Timing Results

In Table 8 the time taken from each layer in 100 MHz frequency is shown, it easily to be noticed that conv1 layer is the bottleneck layer because of large number of input image dimensions(224*224 pixels) and large number of output filters (64 filter).

Table 8: Time for each layer in Squeeze-Net

Layer	Time (µs)
Conv1 & pooling1	1021.725
Fire2	255.805
Fire3 & pooling2	511.325
Fire4	260.765
Fire5 & pooling3	521.245
Fire6	199.955
Fire7	294.045
Fire8	397.075
Fire9	522.525
Conv10	32.035
Total	4.016 ms

The high-speed performance of hardware model can be explained by parallelism in addition to the overlapping between layers. Expand layer starts while squeeze layer still running and pooling layer begins while squeeze and Expand haven't finished yet due to FIFO module that enables the start of Expand or pooling layers even with no complete output of squeeze module is cached as explained in chapter 3.

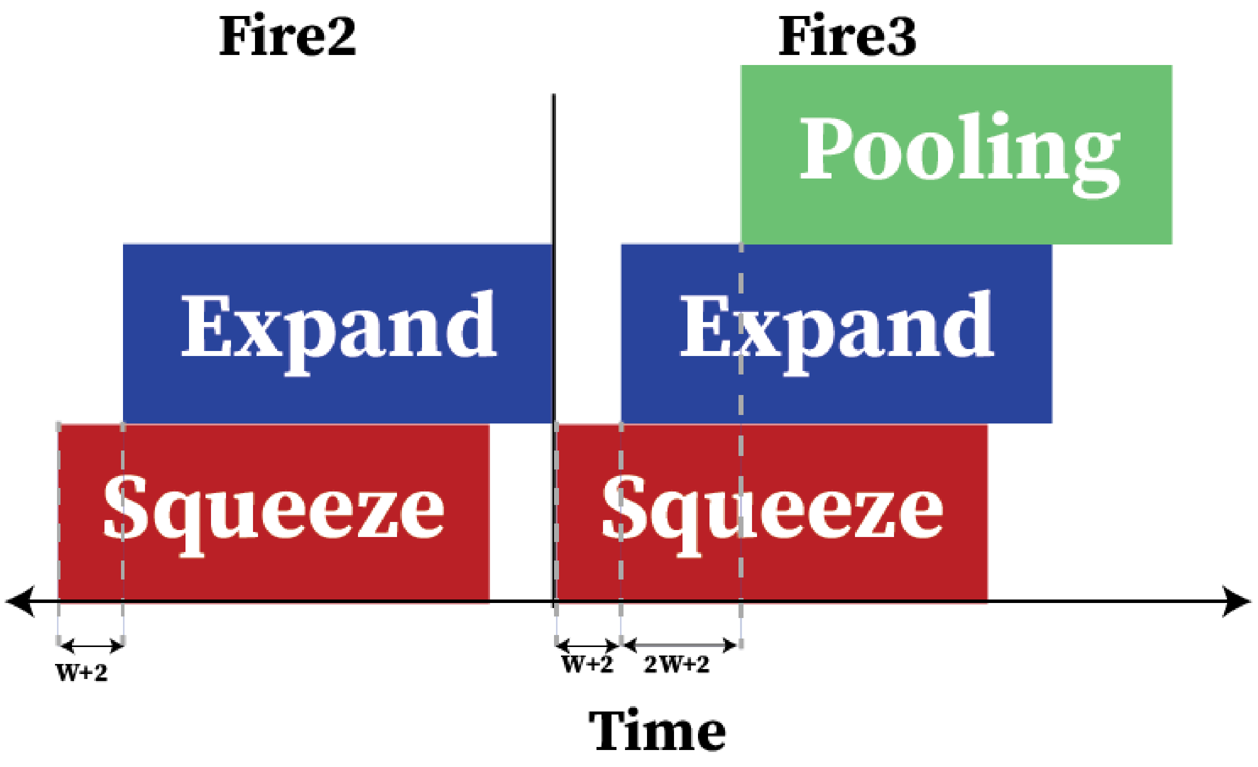


FIGURE 85: OVERLAPPING BETWEEN DIFFERENT LAYERS IN FIRE MODULE

7.3 Power Results

Power consumption is an important metric to analyze the performance of Hardware design compared to GPUs and to decide if it's suitable for many applications or not. Next different methods to reduce the consumed power are discussed.

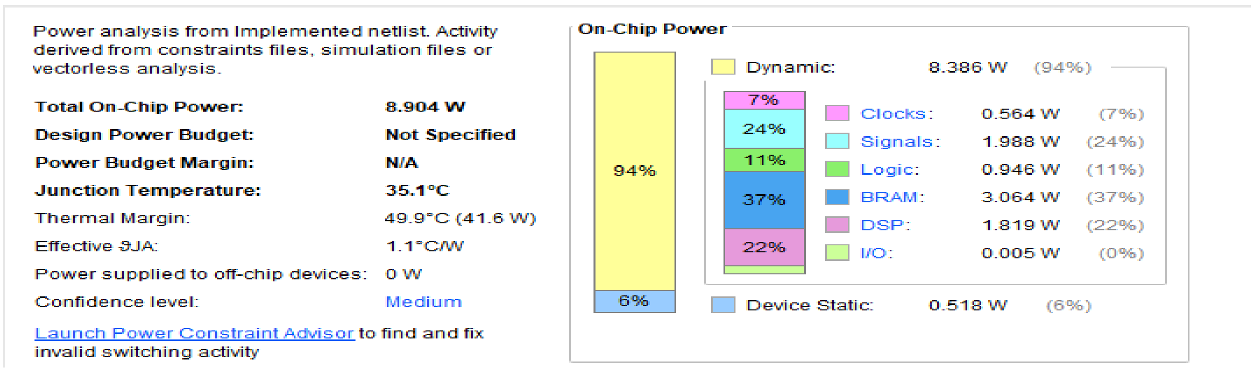
7.3.1 METHODS TO REDUCE POWER CONSUMPTION

- Registering DSP inputs: any arithmetic modules such as adders or multipliers calculate output for every different inputs therefor if input bits from input vectors has different skews, the arithmetic module repeats the operation for every new input until the last delayed bit reaching and the output is settled. Registering inputs make input source is very close to DSP and skew can be neglected and that helps a lot to reduce the switching power.

- Enable signal for FIFO input: let's take fire2 as example, the input for squeeze layer is 64 channel and in every clock cycle squeeze module can perform operations only on 16 channels therefore the desired output is ready after accumulation during 4 clock cycles. Instead of depending on overwriting the values on FIFO input, using an enable signal that high every 4 cycles in fire2 stage can help in reducing the switching power.

- Enable signal for Data memory input: same idea as discussed in pervious point. Write enable signal that is high after the desired output of Expand layer is ready can reduce a lot the writing rate in block RAM.

7.3.2 POWER REPORT SUMMARY



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**8.904 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**35.1°C**
Thermal Margin:	49.9°C (41.6 W)
Effective θJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

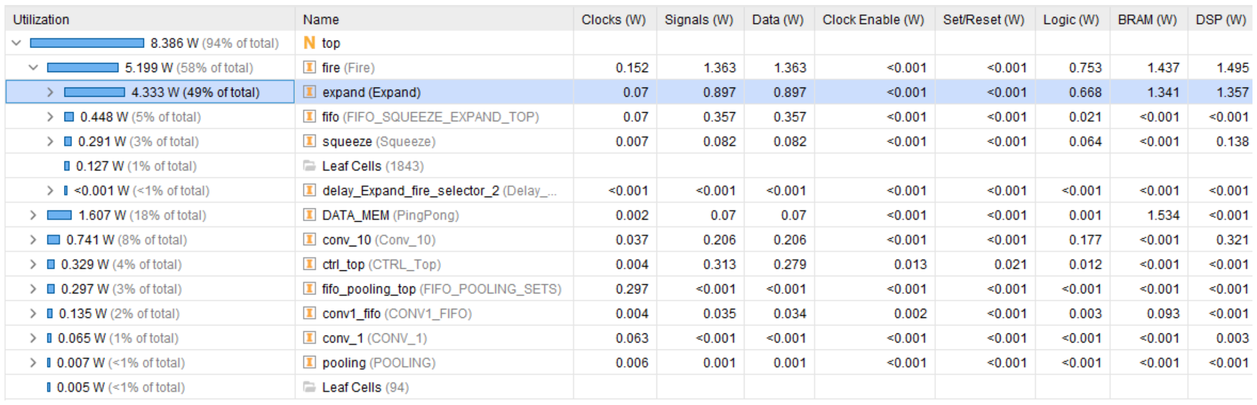
On-Chip Power

Dynamic:	8.386 W	(94%)	
	Clocks:	0.564 W	(7%)
	Signals:	1.988 W	(24%)
	Logic:	0.946 W	(11%)
	BRAM:	3.064 W	(37%)
	DSP:	1.819 W	(22%)
	I/O:	0.005 W	(0%)
Device Static:	0.518 W	(6%)	

FIGURE 86: POWER REPORT SUMMARY FROM VIVADO TOOL

The total on-chip power in proposed design is 8.904 watt. as shown in figure 86, BRAM and DSP consumed around 60% (5 watt) of power dissipated, therefore most of power reduction efforts are focused on those components.

This result is obtained at frequency equals 100 MHz Power and timing has a tradeoff relationship. Higher frequency means higher switching rate and higher power consumption therefore Energy which is Time*Power maybe a more accurate metrics to describe the performance of the design.



Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Set/Reset (W)	Logic (W)	BRAM (W)	DSP (W)
8.386 W (94% of total)	top								
5.199 W (58% of total)	fire (Fire)	0.152	1.363	1.363	<0.001	<0.001	0.753	1.437	1.495
4.333 W (49% of total)	expand (Expand)	0.07	0.897	0.897	<0.001	<0.001	0.668	1.341	1.357
0.448 W (5% of total)	fifo (FIFO_SQUEEZE_EXPAND_TOP)	0.07	0.357	0.357	<0.001	<0.001	0.021	<0.001	<0.001
0.291 W (3% of total)	squeeze (Squeeze)	0.007	0.082	0.082	<0.001	<0.001	0.064	<0.001	0.138
0.127 W (1% of total)	Leaf Cells (1843)								
<0.001 W (<1% of total)	delay_Expand_fire_selector_2 (Delay_...	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
1.607 W (18% of total)	DATA_MEM (PingPong)	0.002	0.07	0.07	<0.001	<0.001	0.001	1.534	<0.001
0.741 W (8% of total)	conv_10 (Conv_10)	0.037	0.206	0.206	<0.001	<0.001	0.177	<0.001	0.321
0.329 W (4% of total)	ctrl_top (CTRL_Top)	0.004	0.313	0.279	0.013	0.021	0.012	<0.001	<0.001
0.297 W (3% of total)	fifo_pooling_top (FIFO_POOLING_SETS)	0.297	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
0.135 W (2% of total)	conv1_fifo (CONV1_FIFO)	0.004	0.035	0.034	0.002	<0.001	0.003	0.093	<0.001
0.065 W (1% of total)	conv_1 (CONV_1)	0.063	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003
0.007 W (<1% of total)	pooling (POOLING)	0.006	0.001	0.001	<0.001	<0.001	<0.001	<0.001	<0.001
0.005 W (<1% of total)	Leaf Cells (94)								

FIGURE 87: THE POWER CONSUMED BY EACH MODULE FROM VIVADO TOOL

As shown in figure 87, expand is the most consumed module and that can be explained as it has the major number of DSPs and weights ROMs which have a very high reading rate. Second module is the data memory.

7.4 OTHER WORKS

In this table we compare our work, 16-bit representation with dynamic clipped-RELU activation function, with the latest works that either implement the whole Squeeze-Net on different FPGAs or implement one of CNNs but with only 10 classes for a certain application, as shown in table 10.

Table 9: Comparison with different implementations for CNNs on FPGA

Design	[22]	[23]	[24]	[25]	This implementation
FPGA	De-10 board	Zynq-7020	De0-NANO board	Artix7 XC7A200T	Virtex-7 VC709
CNN	Squeeze-Net	Squeeze-Net	VGG-Net	Alex-Net	Squeeze-Net
No. of Classes	1000	1000	10	10	10
Frequency (MHz)	100	-	10	100	100
Power(watt)	2	7.95	-	1.5	8.9
Time (ms)	110	1030	21.84	4062	4.02
Energy (mJ)	220	8,188	-	6,093	35.78
Accuracy	55%	57.5%	96%	0% (Over fitted)	66.8%
Utilization BRAMS	-	80%	-	38.22%	67.48%
Utilization DSPs	-	95%	-	16%	73.8%
Utilization FF	-	48%	-	38.72%	18.44%
Utilization LUTs	-	102%	27%	68.66%	22.39%

As shown in the previous table, our implementation requires the highest power that is equal 8.9 watts, however its latency is the smallest between the other implementations and the energy is the smallest as well.

While in the following table, a comparison between our implementation on FPGA Virtex-7 VC709 with 2 different operating frequencies, 100MHz and 172 MHz, and our design implemented on GPU.

As in table 11, GPU requires much more power compared with the implementation on FPGA with both operating frequencies, but its latency is smaller than the implementation on FPGA with operating frequency 100MHz.

The accuracy is same in all cases as shown in table 11 because we retrained the model with the new activation function, Dynamic clipped RELU, and extracted the new weights of all the layers and stored them in our on-chip memory.

Table 10: Comparison with implementation on GPU GeForce RTX 2080 TI

	This Implementation	This implementation	GPU GeForce RTX 2080TI
Frequency	100 MHz	172 MHz	1.545 GHz
Latency (msec)	4.02	2.34	3.02
Power (Watt)	8.9	17.4	55
Accuracy	69.9%	69.9%	69.9%

7.5 SUMMARY

This chapter is discussion about the final results from hardware simulation phase and the summary of power and timing with comparison with pervious works. In next chapter the different methods to enhance the performance of this design are shared.

CHAPTER 8. FUTURE WORK

In this chapter, extra modifications that can be done to implement Squeeze-Net more efficiently with reduction in used clock cycles and FPGA resources will be presented.

8.1 WEIGHT CACHE

Weights of all Fire layers are stored together in the same Bram according to the structure shown in section 2.2.1 However, during each stage we only need a small part of what is stored in the Bram and this part range from 8 to 128 entries out of 1024 location in a single Bram Therefore huge power is consumed to access these deep memories for only few entries.

Shallow memories implemented using LUTs consume less power compared to BRAMs. Therefore, in order to reduce Bram power consumption, we suggest adding a LUT cache of depth 128 that contains the weights of the current processed Fire layer, as shown in figure 87. This way the BRAMs will be accessed one time per fire layer to load the weights in the cache, and the cache will be used instead for processing.

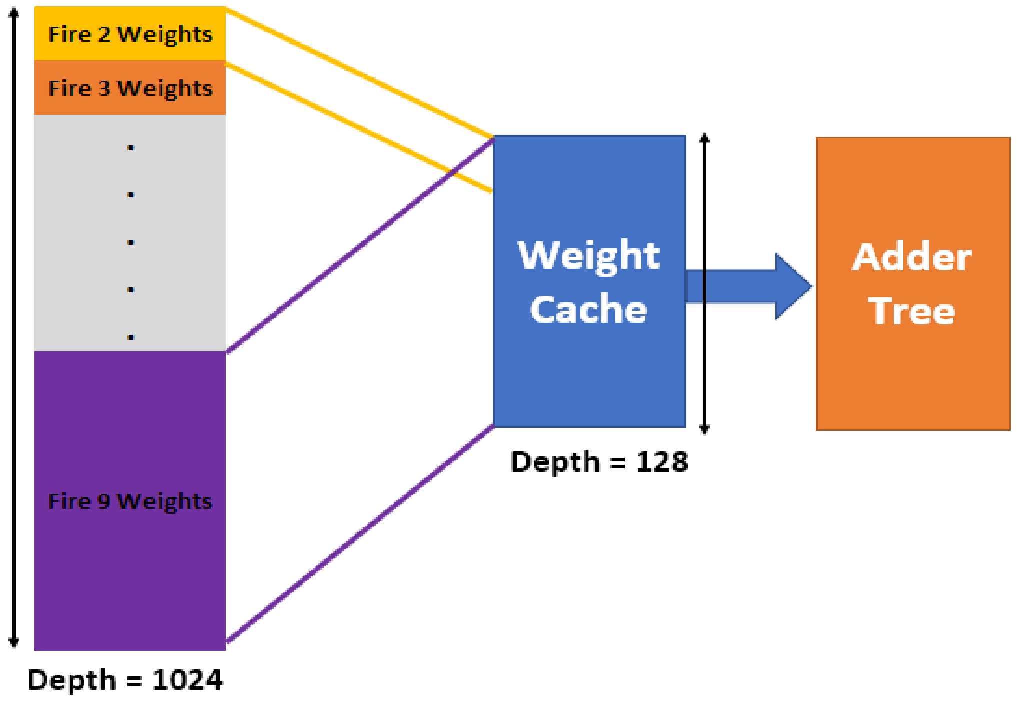


FIGURE 87: CACHE FOR WEIGHTS

8.2 Data cache

Similar to the point 1, the pixel to be processed is re-read several times for different filters from a very deep memory (PING-PONG). Therefore, it's better to store the pixel in a cache and process it from there to reduce power consumption.

8.3 Pipelining Conv 1 and Fire layers

The current design works by finishing Conv 1 and Pool 1 and storing their results in PING-PONG memory and then starting the fire stage. A better solution is to Pipeline Conv 1, Pool 1, and Fire 2 altogether, as shown in figure 88. This way the latency of the design will reduce as shown in Fig#, and the power consumption will reduce too since storing Pool 1 output and reading it for Fire 2 is now removed as the data will enter Fire 2 directly. This will remove about 400K memory write and read operations.

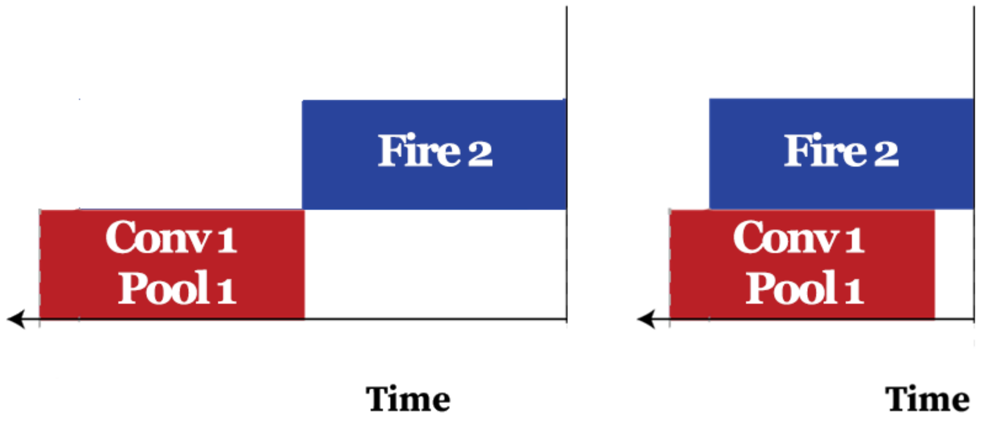


FIGURE 88: NON-OVERLAPPING/ OVERLAPPING BETWEEN CONV-1 AND FIRE2

8.4 Pipelining Fire 9 and Conv 10

Conv10 gets the data directly from fire 9 instead of saving it in data mem to reduce the sharing resources depending on data memory as it is shared between the whole design which is causing congestion and to save clock cycles and power.

8.5 FIFO POOLING REDESIGN

Redesign for module FIFO pooling as 9- Shift register get data and do pooling on them and then start another convolution layer instead of waiting for conv1 to end its operation and store in data mem [to save more utilization and reduce congestion as this module is shared between conv1 and the fires.

REFERENCES:

[1] Asifullah Khan1, 2*, Anabia Sohail1, Umme Zahoora1, and Aqsa Saeed Qureshi1. A Surve y of the
Recent Architectures of Deep Convolutional Neural Networks.

[2]https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/?fbclid=IwAR2qtgvlu1KBFMx5nZ0YXIETZL3RsOEqU8zkrj3IG6L9zqv1UHjPvoBIMy0

[3] Krizhevsky, Alex et al. ImageNet Classification with Deep Convolutional Neural Networks. C ommun.
ACM 60 (2012): 84-90.

[4] https://github.com/albanie/convnet-burden

[5] Simonyan, K. & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image
Recognition. *CoRR*, abs/1409.1556.

[6] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning so me deep
representations.

[7] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network.

[8] Christian Szegedy , Wei Liu, Yangqing Jia, Pierre Sermanet , Scott Reed, Dragomir Anguelov, Dumitru
Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going deeper with convolutions.

[9] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The Pascal Visua l Object
Classes (VOC) Challenge.

[10]] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks.

[11]] Rupesh Kumar Srivastava, Klaus Greff, Jürgen Schmidhuber. Highway Networks.

[12] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dall y, Kurt
Keutzer.SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size.

[13] P. Gysel, M. Motamedi, and S. Ghiasi, “Hardware-oriented Approximation of Convolutional Neural Networks,”

[14] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. “Deep Learning with Limited N umerical Precision”.

[15] M. Motamedi, P. Gysel, V. Akella and S. Ghiasi, "Design space exploration of FPGA-based Deep Convolutional Neural Networks,

[16] Altera Stratix 10 Website. https://www.altera.com/products/fpga/stratix-series/stratix10/overview.html

[17] E. Nurvitadhi et al., "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?”, 2017.

[18] Li, H., & Zhang, Z., & Yang, J., & Liu, L., & Wu, N. (Nov. 6, 2015). A novel vision chip architecture for image recognition based on convolutional neural network. IEEE 2015 International Conference on ASIC (ASICON), 11th. doi: 10.1109/ASICON.2015.7516878.

[19] Ma, Y., & Suda, N., & Cao, Y., & Seo, J., & Vrudhula, S. (Sept. 29, 2016). Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. International 180 Conference on Field Programmable Logic and Applications (FPL), 26th, Session S5bCompilation. doi:10.1109/FPL.2016.7577356

[20] Lacey, G., & Taylor, G., & Areibi, S., (Feb. 13, 2016). Deep Learning on FPGAs: Past, Present, and Future. Cornell University Library. https://arxiv.org/abs/1602.04283

[21] V. Sze et al., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", 2017.

[22] Kathirgamaraja Pradeep, Kamalakkannan Kamalavasan and Ratnasegar Natheesan, EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA.

[23] Meghas Arora and Samyukta Lanka, Accelerating SqueezeNet on FPGA, https://lankas.github.io/15-618Project/?fbclid=IwAR0KgaJF56hJy4K6orrKBU2vdVI7hmsS4uLWC4xHVYx1VujILmUrxI0POCU.

[24] Roman A. Solovyev, Alexandr A. Kalinin, Alexander G. Kustov, Dmitry V. Telpukhov, and Vladimir S. Ruhlov, FPGA Implementation of Convolutional Neural Networks with Fixed-Point Calculations.

References

[25] Mark A. Espinosa, IMPLEMENTATION OF CONVOLUTIONAL NEURAL NETWORKS IN FPGA FOR IMAGE CLASSIFICATION.

[26] XILINX, Vivado Design SuiteUser Guide Programming and Debugging UG908 (v2018.2) June 6, 2018.

[27] XILINX, VC709 Evaluation Board for the Virtex-7 FPGA User Guide UG887 (v1.6) March 11, 2019.

[28] XILINX, Vivado Design Suite User Guide Using Constraints UG903 (v2018.3) December 5, 2018.

[29] XILINX, VC709 Si570 Programming October 2013.

[30] XILINX, Vivado Design Suite User Guide Synthesis UG901 (v2019.1) June 12, 2019.

[31] XILINX, Vivado Design Suite User Guide Release Notes, Installation, and Licensing UG973 (v2015.4) November 18, 2015.

[32] XILINX, Vivado Design Suite User Guide: Logic Simulaton UG900 (v2019.2) October 30, 2019.