Cairo University

LOW POWER HARDWARE IMPLEMENTATION OF CONVOLUTIONAL NEURAL NETWORK

Under the Supervision of

Dr. Hassan Mostafa

A 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

By

Ibrahim Medhat Tawfik Essmat

Amr Gamal El-Sayed Attia

Mohamed Ayman Youssef Ahmed

Mahmoud Abdel-Halim El-Sayed

Mennatullah Sayed Abbas Younis

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
July - 2018

Acknowledgments

We are very grateful to have worked with many wonderful people throughout our graduation project. First and foremost, we would like to express our sincere gratitude to our advisor, Assistant Professor Hassan Mostafa. This thesis would not have been a success without his guidance and support. We would like also to thank ONE Lab's RAs for their great support.

Table of Contents

List of Figures

Abstract

The convolutional neural network (CNN) has found wide acceptance in solving practical computer vision and image recognition problems. Also recently, due to its flexibility, faster development time, and energy efficiency, the field-programmable gate array (FPGA) has become an attractive solution to exploit the inherent parallelism in the feed-forward process of the CNN. However, to meet the demands for high accuracy of today's practical recognition applications that typically have massive datasets; the sizes of CNNs have to be larger and deeper. Enlargement of the CNN aggravates the problem of off-chip memory bottleneck in the FPGA platform since there is not enough space to save the pretrained network large parameters on-chip. This thesis proposes an FPGA-based low power solution for CNN inference which is used in image classifications. The implemented architecture is AlexNet network, which consists of five convolutional layers and three fully connected layers, same hardware is used to fit all layers and to minimize the resources used on the FPGA.

Chapter 1 : Introduction

1.1. Motivation

Artificial intelligence (AI) is the general science and engineering principle which deals with how to create intelligent machines. A subgroup within AI, machine learning algorithms learn from large quantities of statistical data and make predictions on unknown data. Since the 1980s, algorithmic approaches in decision tree learning, inductive logic programming, clustering, reinforcement learning, and Bayesian networks resulted in many new technical advances and solutions for real-world problems, such as spam filtering, optimal control, image and speech processing, and many others. Within machine learning, deep learning is a new branch of algorithm which consists of multiple non-linear layers.

Extracting semantic contents from images are tasks that have attracted a lot of research interest. Although recognizing and category images seems like a simple task, more sophisticated datasets require a robust classifier that can handle the big data demands of 21st century, researchers are still looking for larger, faster and more economical solutions. Designing a system can extract the relevant information from an image efficiently requires computer vision and machine learning.

There are some computer vision systems have successfully closed the gap with human performance, such as handwritten digit recognition [1]. However, today's advanced classification systems encounter problems when there are a large number of objects or high similarity between these objects. For instance, the ImageNet dataset has several thousand categories, and it is very hard to get the right categories if the difference are more subtle in terms of variety or type. Examples of such cases are recognize different car models or different kinds of dogs.

1.2. Previous Work

Most previous work on CNN accelerators mainly focused on computation engine optimization without considering the memory bandwidth limitation effect on the engine throughput. The work by Peemen et al., however, took the opposite approach and optimized just the memory bandwidth. Hence, these accelerators are therefore either communication- or computation-bounded. This problem was solved by Zhang et al., where they used the Rooine analytical model to match the memory bandwidth with the computation throughput. The work by Zhang et al. can be further improved since they underutilized the available bandwidth - using only 2.2 GB/s out of 4.5 GB/s bandwidth available - by running the computation engine at 100 MHz.

Energy efficiency is also becoming more crucial, especially in the real-time recognition of objects in power-constrained autonomous systems and robots [4]. In an effort to design a CNN hardware accelerator that is energy-efficient, Zhang C proposed an FPGA cluster to accelerate each layer of the network independently. However, drawbacks of the work are the large board-to-board communication overhead and additional energy utilization of an extra FPGA solely used for the purpose of controlling the remaining FPGAs.

There are two well-established methods to save on energy in embedded devices, and they are dynamic voltage and frequency scaling (DVFS) and the RTH mechanism. Integrated circuits are recently equipped with a number of sleep-state features. These features are mainly based on RTH, which can be used to optimize the energy efficiency of the system. RTH enables the system to execute the task at high frequency, hence completing it quickly and allowing the hardware to be turned off earlier. DVFS, on the contrary, decreases the operating frequency and therefore allows the circuit to operate at a lower voltage. Consequently, DVFS minimizes dynamic energy dissipation with the sacrifice of static energy loss while RTH does completely the reverse.

1.3. Background

1.3.1. Artificial Intelligence and DNNs

DNNs, also referred to as deep learning, are a part of the broad field of AI, which is the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do, according to John McCarthy, the computer scientist who coined the term in the 1950s. The relationship of deep learning to the whole of artificial intelligence is illustrated in (Fig).

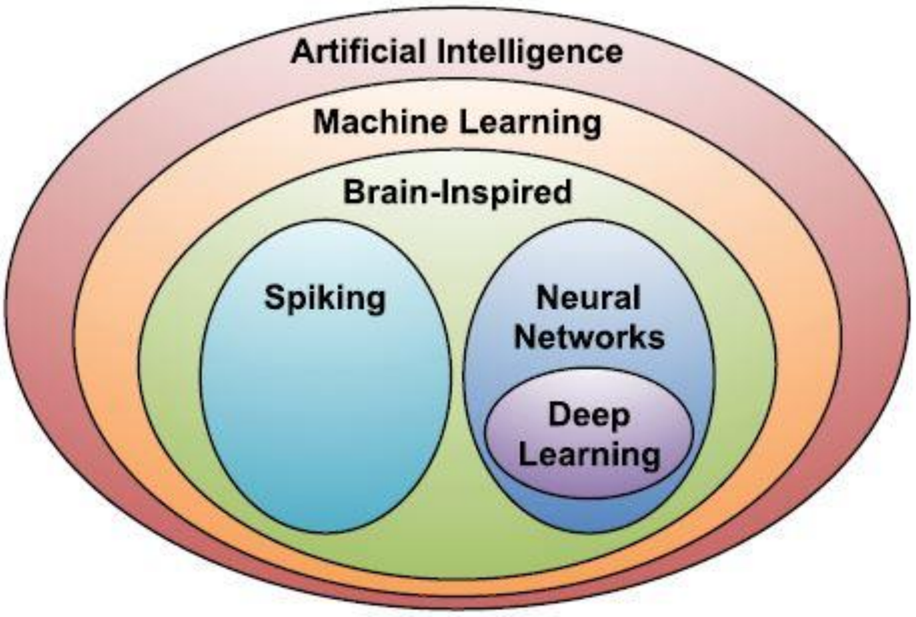


Figure 1. AI relation with Deep Learning

Within AI is a large subfield called machine learning, which was defined in 1959 by Arthur Samuel as the field of study that gives computers the ability to learn without being explicitly programmed. That means a single program, once created, will be able to learn how to do some intelligent activities outside the notion of programming. This is in contrast to purpose-built programs whose behavior is defined by hand-crafted heuristics that explicitly and statically define their behavior. The advantage of an effective machine learning algorithm is clear. Instead of the laborious and hit-or-miss approach of creating a distinct, custom program to solve each individual problem in a domain, the single machine learning algorithm simply needs to learn, via a process called *training*, to handle each new problem. Within the machine learning field, there is an area that is often referred to as brain-inspired computation. Since the brain is currently the best “machine” we know for learning and solving problems, it is a natural place to look for a machine learning approach. Therefore, a brain-inspired computation is a program or algorithm that takes some aspects of its basic form or functionality from the way the brain works. This is in contrast to attempts to create a brain, but rather the program aims to emulate some aspects of how we understand the brain to operate. Although scientists are still exploring the details of how the brain works, it is generally believed that the main computational element of the brain is the *neuron*. There are approximately 86 billion neurons in the average human brain. The neurons themselves are connected together with a number of elements entering them called dendrites and an element leaving them called an axon. The neuron accepts the signals entering it via the dendrites, performs a computation on those signals, and generates a signal on the axon. These input and output signals are referred to as *activations*. This will be further illustrated in the upcoming chapters.

1.4. Organization of the thesis

The thesis is organized as follows; chapter 2 discusses the neural networks and different types of neural networks, chapter 3 discusses the software aspects concerning the software tool on which the network was developed, benchmark datasets and FPGAs versus GPUs. Chapter 4 discusses the hardware implemented design including the FPGA utilization, clock frequency, number of MAC operations, etc.., chapter 5 discusses the results obtained and a brief discussion of their significance. Finally, chapter 6 discusses the future work and the improvements that could be made on the current design to enhance its performance.

Chapter 2 : Neural Networks

2.1. Introduction

Neural networks are a popular type of machine learning model. They consist of an ordered set of layers, where every layer is a set of nodes. The first layer of the neural network is called the input layer, and the last one is called the output layer. A neural network with only one layer of synaptic weights can only approximate linear functions of the inputs. Therefore, intermediate layers are introduced to form more powerful neural networks in order to approximate more complex models. These intermediate layers are known as hidden layers since their states are usually not directly observable. Neural networks with one or more hidden layers are referred to as deep neural networks

Layers are a semantic group of nodes. Nodes belonging to one layer are connected to the nodes in the following and/or the previous layers. These connections are weighted edges, and they are referred to as weights.

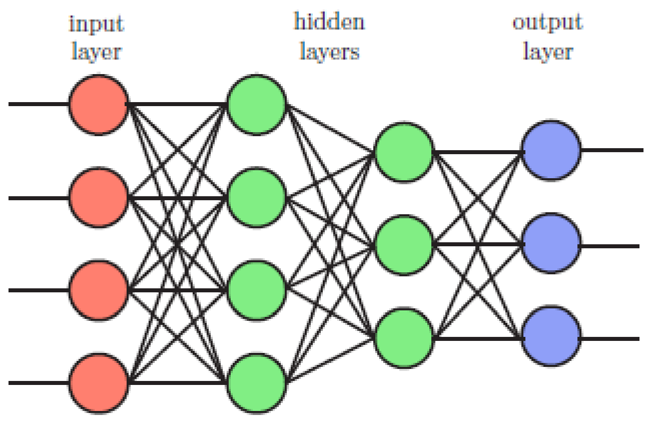


Figure 2. Neural Network

A special case of a neural network called the convolutional neural network (CNN) is the primary focus of this thesis. Before discussing CNNs, we will discuss how regular neural networks work and talk about the ANNs.

2.2. Neurons

The neuron is the basic element in an artificial neural network (Figure 2.1). A neuron receives inputs that may come from the observable properties of a given problem (e.g., image pixels, or audio samples) or may be the outputs of other neurons. Each connection from an input to a neuron is associated with a synaptic weight. The neuron sums up the products of all pairs of inputs and synaptic weights. This weighted sum is typically offset with a bias term in order to make the model more general. The bias term can be considered as an additional synaptic weight that is always connected to a constant input of +1.

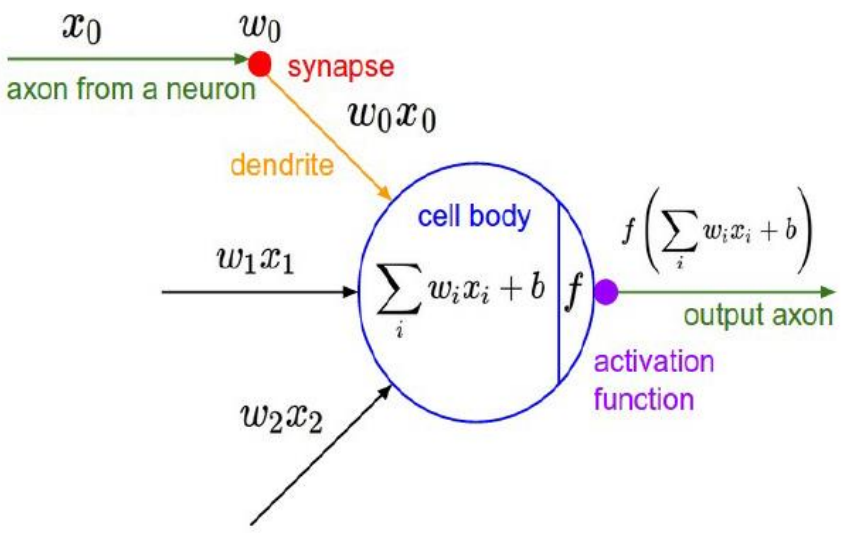


Figure 3. Mathematical model of a neuron

2.2.1. Activation Function

Nonlinearity is introduced into neural networks by introducing an activation function at the neuron output. The activation function transfers the weighted sum to an output value that defines the state of the neuron.

Example activation functions may be a sigmoid function that maps weighted sum values from (-∞ , +∞) to (0, 1) , or a Rectified Linear Unit (ReLU) which clamps all negative values to 0 and retains all positive values (Figure ----) and it is very commonly used.

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

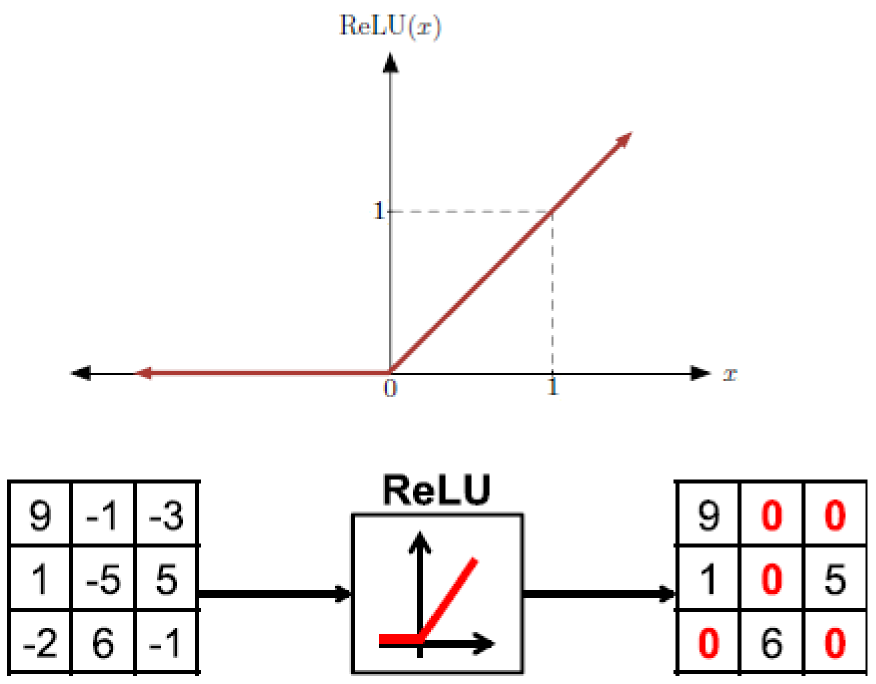


Figure 4. ReLU activation function

ReLU leads to sparsity. As a result, given an input, only a subset of nodes are non-zero (active) and every possible subset results with a linear function.

2.3. Inference Versus Training

Since DNNs are an instance of a machine learning algorithm, the basic program does not change as it learns to perform its given tasks. In the specific case of DNNs, this learning involves determining the value of the weights (and bias) in the network, and is referred to as *training* the network. Once trained, the program can perform its task by computing the output of the network using the weights determined during the training process. Running the program with these weights is referred to as *inference*. In this section, we will use image classification, as shown in (Fig.), as a driving example for training and using a DNN.

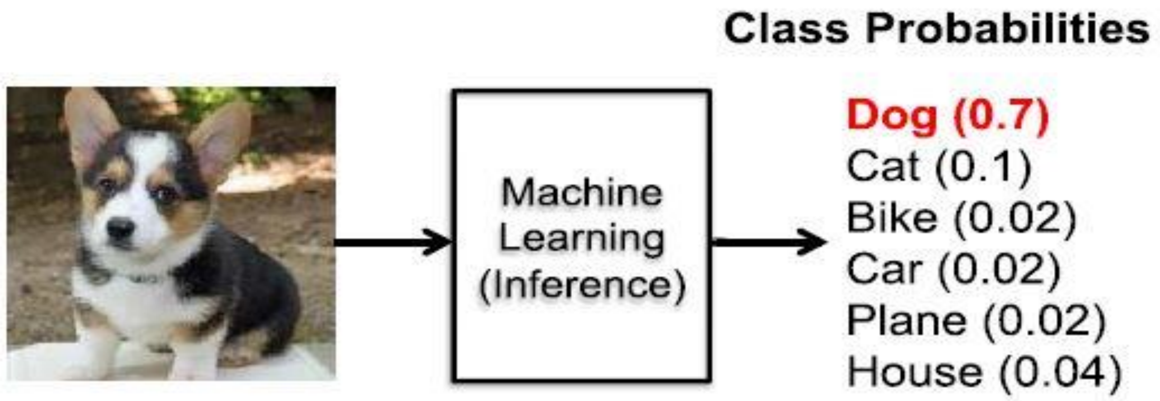


Figure 5. Class detection

When we perform inference using a DNN, we give an input image and the output of the DNN is a vector of scores, one for each object class; the class with the highest score indicates the most likely class of object in the image. The overarching goal for training a DNN is to determine the weights that maximize the score of the correct class and minimize the scores of the incorrect classes. When training the network the correct class is often known because it is given for the images used for training (i.e., the training set of the network). The gap between the ideal correct scores and the scores computed by the DNN based on its current weights is referred to as the *loss* (L). Thus the goal of training DNNs is to find a set of weights to minimize the average loss over a large training set. When training a network, the weights (w_{ij}) are usually updated using a hill-climbing optimization process called gradient descent. A multiple of the gradient of the loss relative to each weight, which is the partial derivative of the loss with respect to the weight, is used to update the weight (i.e., updated $W_{ij}^{t+1} = W_{ij}^{t} +$ where α is called the learning rate). Note that this gradient indicates how the weights should change in order to reduce the loss. The process is repeated iteratively to reduce the overall loss.

An efficient way to compute the partial derivatives of the gradient is through a process called *backpropagation*. Backpropagation, which is a computation derived from the *chain rule* of calculus, operates by passing values backwards through the network to compute how the loss is affected by each weight. This backpropagation computation is, in fact, very similar in form to the computation used for inference as shown in (Fig).Thus, techniques for efficiently performing inference can sometimes be useful for performing training. It is, however, important to note a couple of points. First, backpropagation requires intermediate outputs of the network to be preserved for the backwards computation, thus training has increased storage requirements. Second, due to the gradients use for hill climbing, the precision requirement for training is generally higher than inference. A variety of techniques are used to improve the efficiency and robustness of training. For example, often the loss from multiple sets of input data, i.e., a *batch*, are collected before a single pass of weight update is performed; this helps to speed up and stabilize the training process. There are multiple ways to train the weights. The most common approach, as described above, is called *supervised learning*, where all the training samples are labeled (e.g., with the correct class). *Unsupervised learning* is another approach where all the training samples are not labeled and essentially the goal is to find the structure or clusters in the data. *Semi-supervised learning* falls in between the two approaches where only a small subset of the training data is labeled (e.g., use unlabeled data to define the cluster boundaries, and use the small amount of labeled data to label the clusters). Finally, *reinforcement learning* can be used to the train weights such that given the state of the current environment, the DNN can output what action the agent should take next to maximize expected rewards; however, the rewards might not be available immediately after an action, but instead only after a series of actions. Another commonly used approach to determine weights is *fine-tuning*, where previously trained weights are available and are used as a starting point and then those weights are adjusted for a new data set (e.g., transfer learning) or for a new constraint (e.g., reduced precision). This results in faster training than starting from a random starting point, and can sometimes result in better accuracy. This article will focus on the efficient processing of DNN inference rather than training, since DNN inference is often performed on embedded devices (rather than the cloud) where resources are limited as discussed in more details later.

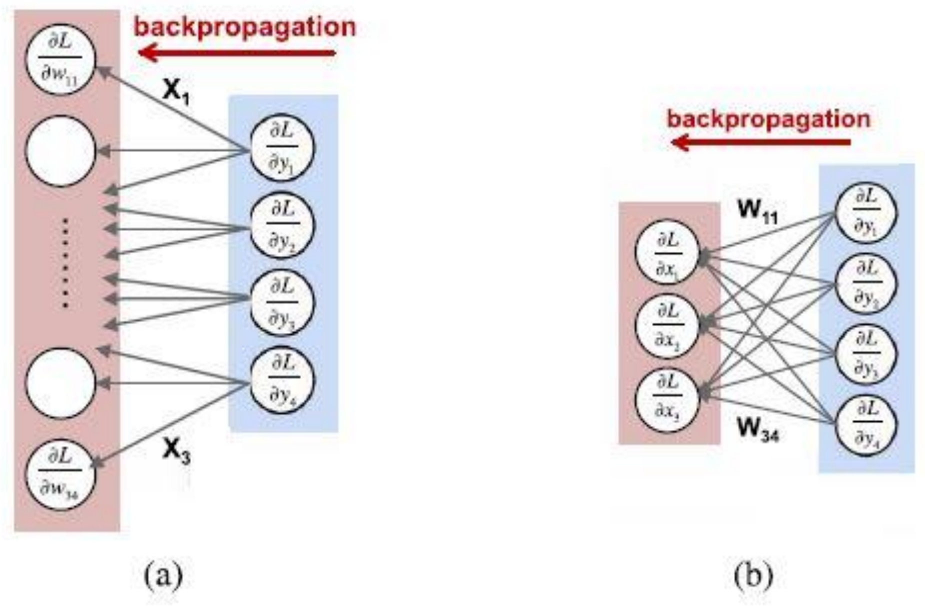

backpropagation
X_1
X_3
$\frac{\partial L}{\partial w_{11}}$
$\frac{\partial L}{\partial w_{34}}$
$\frac{\partial L}{\partial y_1}$
$\frac{\partial L}{\partial y_2}$
$\frac{\partial L}{\partial y_3}$
$\frac{\partial L}{\partial y_4}$
(a)
backpropagation
W_{11}
W_{34}
$\frac{\partial L}{\partial x_1}$
$\frac{\partial L}{\partial x_2}$
$\frac{\partial L}{\partial x_3}$
$\frac{\partial L}{\partial y_1}$
$\frac{\partial L}{\partial y_2}$
$\frac{\partial L}{\partial y_3}$
$\frac{\partial L}{\partial y_4}$
(b)

Figure 6. Backpropagation algorithm

2.4. Embedded Versus Cloud

The various applications and aspects of DNN processing (i.e., training versus inference) have different computational needs. Specifically, training often requires a large data sets and significant computational resources for multiple weight-update iterations. In many cases, training a DNN model still takes several hours to multiple days and thus is typically performed in the cloud. Inference, on the other hand, can happen either in the cloud or at the edge (e.g., IoT or mobile).

In many applications, it is desirable to have the DNN inference processing near the sensor. For instance, in computer vision applications, such as measuring wait times in stores or predicting traffic patterns, it would be desirable to extract meaningful information from the video right at the image sensor rather than in the cloud to reduce the communication cost. For other applications such as autonomous vehicles, drone navigation, and robotics, local processing is desired since the latency and security risks of relying on the cloud are too high. However, video involves a large amount of data, which is computationally complex to process; thus, low cost hardware to analyze video is challenging yet critical to enabling these applications. Speech recognition enables us to seamlessly interact with electronic devices, such as smartphones. While currently most of the processing for applications such as Apple Siri and Amazon Alexa voice services is in the cloud, it is still desirable to perform the recognition on the device itself to reduce latency and dependency on connectivity, and to improve privacy and security.

Many of the embedded platforms that perform DNN inference have stringent energy consumption, compute and memory cost limitations; efficient processing of DNNs has thus become of prime importance under these constraints. Therefore, in this thesis, we will focus on the compute requirements for inference rather than training.

2.5. DNN Applications

Many applications can benefit from DNNs ranging from multimedia to medical space. In this section, we will provide examples of areas where DNNs are currently making an impact and highlight emerging areas where DNNs hope to make an impact in the future.

* **Image and video:** Video is arguably the biggest of the big data. It accounts for over 70% of today’s Internet traffic. For instance, over 800 million hours of video is collected daily worldwide for video surveillance. Computer vision is necessary to extract meaningful information from video. DNNs have significantly improved the accuracy of many computer vision tasks such as image classification, object localization and detection, image segmentation, and action recognition.
* **Speech and language:** DNNs have significantly improved the accuracy of speech recognition as well as many related tasks such as machine translation, natural language processing, and audio generation. An impact and highlight emerging areas where DNNs hope to make an impact in the future.
* **Robotics DNNs:** They have been successful in the domain of robotic tasks such as grasping with a robotic arm, motion planning for ground robots, visual navigation, control to stabilize a quadcopter, and driving strategies for autonomous vehicles.

DNNs are already widely used in multimedia applications today (e.g., computer vision, speech recognition). Looking forward, we expect that DNNs will likely play an increasingly important role in the medical and robotics fields, as discussed above, as well as finance (e.g., for trading, energy forecasting, and risk assessment), infrastructure (e.g., structural safety, and traffic control), weather forecasting, and event detection. The myriad application domains pose new challenges to the efficient processing of DNNs; the solutions then have to be adaptive and scalable in order to handle the new and varied forms of DNNs that these applications may employ.

2.6. Popular DNN Models

Many DNN models have been developed over the past two decades. Each of these models has a different "network architecture" in terms of number of layers, layer types, layer shapes (i.e., filter size, number of channels and filters), and connections between layers. Understanding these variations and trends is important for incorporating the right flexibility in any efficient DNN engine. In this section, we will give an overview of various popular DNNs such as LeNet as well as those that competed in and/or won the ImageNet Challenge as shown in Fig. 7.

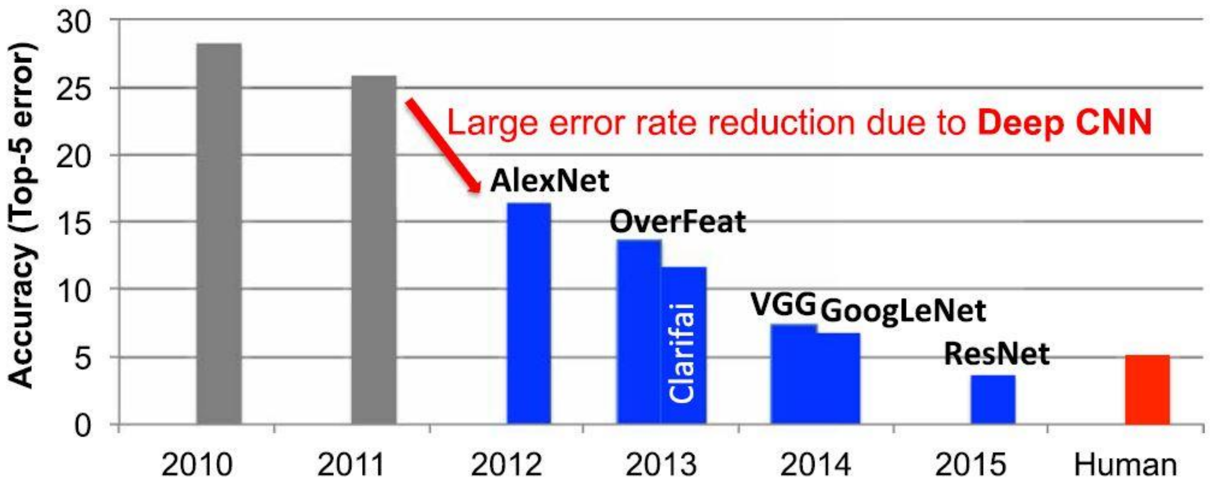
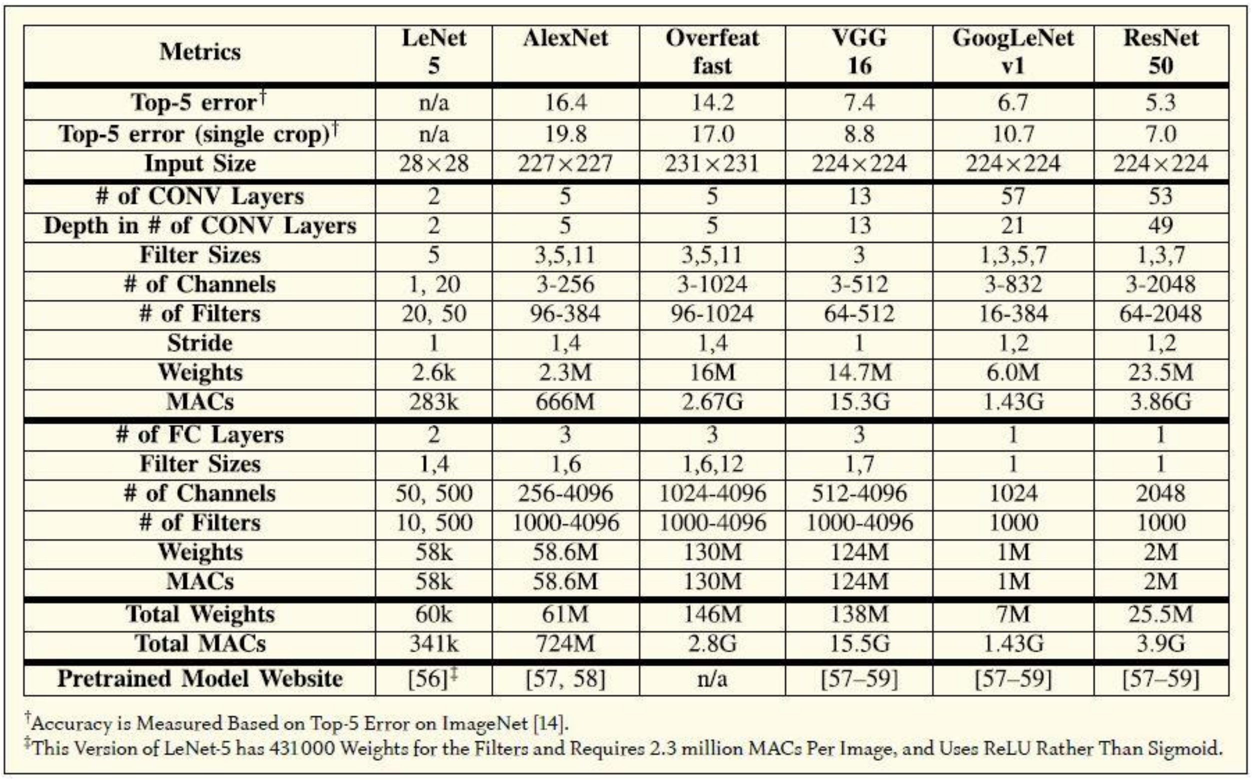


Figure 7. Error Rate Reduction

The DNN models are summarized in (Table). Two results for top-5 error results are reported. In the first row, the accuracy is boosted by using multiple crops from the image and an ensemble of multiple trained models (i.e., the DNN needs to be run several times); these results were used to compete in the ImageNet Challenge. The second row reports the accuracy if only a single crop was used (i.e., the DNN is run only once), which is more consistent with what would likely be deployed in real-time and/or energy-constrained applications. *AlexNet* was the first CNN to win the ImageNet Challenge in 2012. It consists of five CONV layers and three FC layers.

Within each CONV layer, there are 96 to 384 filters and the filter size ranges from 3×3 to 11×11, with 3 to 256 channels each. In the first layer, the three channels of the filter correspond to the red, green, and blue components of the input image. A ReLU nonlinearity is used in each layer. Max pooling of 3×3 is applied to the outputs of layers 1, 2, and 5. To reduce computation, a stride of 4 is used at the first layer of the network. AlexNet introduced the use of LRN in layers 1 and 2 before the max pooling, though LRN is no longer popular in later CNN models. One important factor that differentiates AlexNet from LeNet is that the number of weights is much larger and the shapes vary from layer to layer. To reduce the amount of weights and computation in the second CONV layer, the 96 output channels of the first layer are split into two groups of 48 input channels for the second layer, such that the filters in the second layer only have 48 channels. Similarly, the weights in fourth and fifth layers are also split into two groups. In total, AlexNet requires 61 million weights and 724 million MACs to process one 227×227 input image.



Metrics	LeNet 5	AlexNet	Overfeat fast	VGG 16	GoogLeNet v1	ResNet 50
Top-5 error[†]	n/a	16.4	14.2	7.4	6.7	5.3
Top-5 error (single crop)[†]	n/a	19.8	17.0	8.8	10.7	7.0
Input Size	28×28	227×227	231×231	224×224	224×224	224×224
# of CONV Layers	2	5	5	13	57	53
Depth in # of CONV Layers	2	5	5	13	21	49
Filter Sizes	5	3,5,11	3,5,11	3	1,3,5,7	1,3,7
# of Channels	1, 20	3-256	3-1024	3-512	3-832	3-2048
# of Filters	20, 50	96-384	96-1024	64-512	16-384	64-2048
Stride	1	1,4	1,4	1	1,2	1,2
Weights	2.6k	2.3M	16M	14.7M	6.0M	23.5M
MACs	283k	666M	2.67G	15.3G	1.43G	3.86G
# of FC Layers	2	3	3	3	1	1
Filter Sizes	1,4	1,6	1,6,12	1,7	1	1
# of Channels	50, 500	256-4096	1024-4096	512-4096	1024	2048
# of Filters	10, 500	1000-4096	1000-4096	1000-4096	1000	1000
Weights	58k	58.6M	130M	124M	1M	2M
MACs	58k	58.6M	130M	124M	1M	2M
Total Weights	60k	61M	146M	138M	7M	25.5M
Total MACs	341k	724M	2.8G	15.5G	1.43G	3.9G
Pretrained Model Website	[56][‡]	[57, 58]	n/a	[57–59]	[57–59]	[57–59]

[†]Accuracy is Measured Based on Top-5 Error on ImageNet [14].
[‡]This Version of LeNet-5 has 431 000 Weights for the Filters and Requires 2.3 million MACs Per Image, and Uses ReLU Rather Than Sigmoid.

2.7. Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model that is used for machine learning and pattern recognition. The name and the basic concept are inspired by how the human brain uses a network of neurons to recognize and classify objects.

Depending on the input, different neurons activate (or fire), making the brain able to decide what kind of pattern it is detecting. An ANN can intuitively be viewed as a probabilistic classifier. Depending on the input data, it will calculate the probability that the data belongs to a certain class (e.g. an object in an image or an investment decision). The network can be trained to recognize different classes by being provided a set of labeled training data, e.g. a set of faces and a set of non-faces. It can then learn to decide whether an image contains a face or not. This is called supervised learning. The network can also be trained to be unsupervised, by providing it with a set of unlabeled images. The latter technique is used to find hidden structures in the data, by learning the network to recreate the input. But for this thesis only supervised learning is relevant.

2.7.1. Definition

An ANN consists of a number of layers containing a set of the so-called neurons, also known as units. A neuron takes in a set of values as input (e.g. image pixels), where each value is associated with a respective weight. The input and the weights are multiplied and summed, and the result is used to calculate a non-linear activation function. Formally a neuron's input and output is defined as:

$$Input = \{x_1, x_2, \dots \dots \dots \dots \dots \dots \dots \ x_n\}$$

$$Output = f(w^T x) = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = 0$$

Where **w** is the vector containing the connection weights and **b** is the neuron bias. $f(\sum_{i=1}^{n} w_i x_i + b)$ is the activation function, which emulates the activation of a neuron in the brain, i.e. it decides whether the neuron is on or off. It also causes the values in the network to have a reasonable value interval. $f(\sum_{i=1}^{n} w_i x_i + b)$ tends to be either:

Sigmoid: $f(z) = \frac{1}{1 + e^{-z}}$

Hyperbolic tangent: $f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

2.7.2. Issues with Object Recognition

While the ANN model has proven useful in several applications, it falls short when it comes to object recognition in images. There are three major reasons for this:

1. **Topology**

A fully connected ANN does not take into consideration the topology of the input. An image has a strong 2D spatial locality correlation, which makes it possible to combine low-order features (edges, end-points, etc…) in the same area into higher-order features (noses, ears etc…).

2. **Scalability**

Even small images contains a large amount of pixels/inputs, a 32x32 image contains 1024 pixels/inputs. A fully connected network with 100 hidden units would then end up with 1024x100 weights that need to be calculated in the first layer, Thus making it harder to scale for larger images and rather inefficient.

3. **Object variance**

While objects are similar enough, on a higher level, to be grouped together into a class, they can still be very different on a lower level. e.g. a human face have several features that are needed for it to be defined as a face, e.g. eyes, mouth, nose etc. But the size and shape of these features tend to be very different from person to person. While it is possible for a standard ANN to compensate for these internal differences within a class, it would have to make three costly compensations.

1) The network would have to be very large.

2) It would probably contain several neurons with similar weight vectors positioned at different places in the network.

3) It would require a massive amount of training samples.

2.8. Convolutional Neural Networks

2.8.1. Introduction

The first thing to know about convolutional networks is that they don’t perceive images like humans do. Therefore, you are going to have to think in a different way about what an image means as it is fed to and processed by a convolutional network. Generally speaking, Convolutional Neural Networks (ConvNets) are a class of Neural Networks which are made to process images as input data. CNN architectures vary in the number and type of layers implemented for its specific application. The layers of a ConvNet have neuron organized in 3 dimensions: width, height, depth. So that each layer accepts 3D input and transforms it to a 3D output.

Neurons in each CNN layer are arranged in a 3D arrangement, and transform a three dimensional output from a three dimensional input. For our particular application, the input layer holds the images as 3D inputs, with the height, width and RGB values as dimensions.

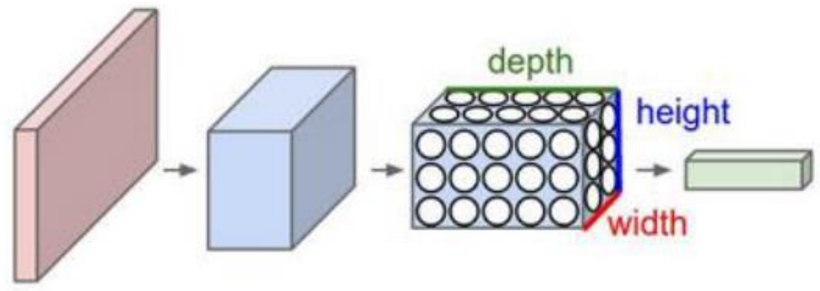


Figure 8. Width, depth and height visualized

Figure --- A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations.

The CNN algorithm is constructed by stacking multiple computational layers for feature extraction and classification. Modern CNNs achieve their superior accuracy by building a very deep hierarchy of layers which transform the input image data into highly abstract representations called feature maps (fmaps). The primary computation in the CNN layers is performing the high-dimensional convolutions. Each layer applies some **kernels**, which are commonly known as **filters**, on the input feature maps (**ifmaps**) to extract the embedded characteristics and generate the output feature maps (**ofmaps**) by accumulating the partial sums (**psums**) generated from the convolution of those multi-depth filters and the input feature maps. Both filters and feature maps are considered to be 4-D: each filter or fmap is a 3-D structure consisting of multiple 2-D planes, i.e., channels, and a batch of 3-D ifmaps is processed by a group of 3-D filters in each layer. In addition, there is a 1-D bias that is added to the filtered results.

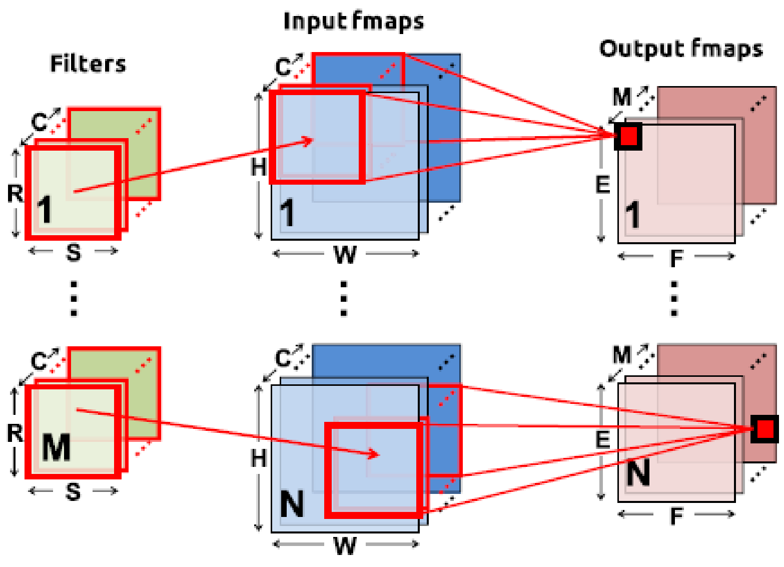


Figure 9. Convolution operation

Shape parameters	Description
N	Batch size of 3D FMAPS
M	# of 3D filters / # of output FMAPS channels
C	# of input FMAPS / filters channels
H/W	Input FMAPS plane height/width
R/S	Filter plane height/width
E/F	Output FMAPS plane height/width

Given the shape parameters in Table I, the computation of a layer is defined as

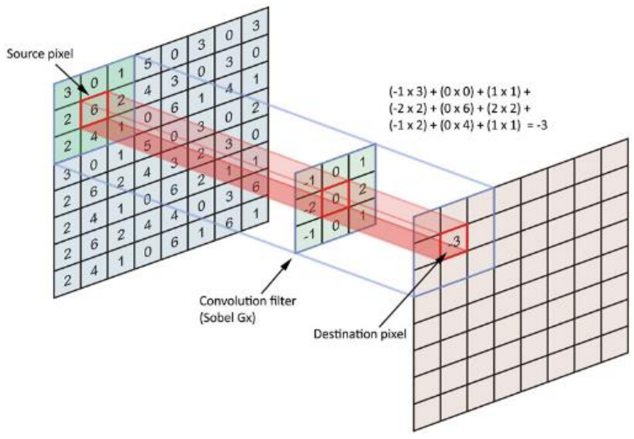
$$O[z][u][x][y] = ReLU\left(B[u] + \sum_{k=0}^{C-1}\sum_{i=0}^{R-1}\sum_{j=0}^{S-1} I[z][k][U_x + i]\left[U_y + j\right] * W[u][k][i][j]\right)$$

Where **O**, **I**, **W**, and **B** are the matrices of the OFMAPS, IFMAPS, filters, and biases, respectively. U is a given stride size. Fig.() shows a visualization of this computation (ignoring biases for simplicity). After the convolutions, activation functions, such as the Rectified Linear Unit (ReLU), are applied to introduce nonlinearity.

This section gives an overview about the major concepts that are applied in convolutional neural networks. Four principle layer types exist that are used to build CNN configurations. Each Layer is explained in one of the following subsections.

2.8.2. Convolutional Layers

This is the core building block of the network. Its parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have a size of 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the colour channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.



Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Figure 10. Convolution visualization

A dot product is a computation between the filters and the input, the ConvNet will learn filters that activate when some specific type of feature in the input is detected.

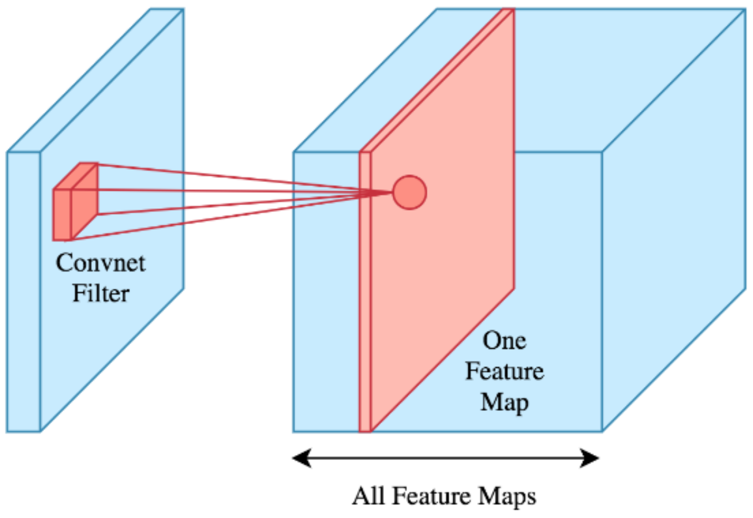


Figure 11. Output feature map computation

Local Connectivity: When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyper parameter called the receptive field of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.

Three hyper parameters control the size of the output volume: the depth, stride and zero-padding. We discuss these next: First, the depth of the output volume is a hyper parameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of colour. We will refer to a set of neurons that are all looking at the same region of the input as a depth column (some people also prefer the term fibre).

Second, we must specify the stride with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.

As we will soon see, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyper parameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as we'll see soon we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same-).

2.8.3. Pooling Layers

It is common to periodically insert a Pooling layer in-between successive Convolution layers in CNN architectures. Just like strides, pooling is another way of reducing the dimensionality of a layer. Similar to the convolution operation, pooling methods also work with windows and strides. But this time, instead of applying a weight, bias and activation function, they apply simpler functions. Depending on the task, one may choose from various pooling methods. Pooling summarizes a K x K area of the input feature map. As a stride of 2 is the usual choice, pooling layers are sometimes also called subsampling layers. Typically, the window dimensions are {2x2, 3x3, 4x4, 5x5}, and a stride of 2. Popular examples of architectures using the pooling layers are AlexNet and VGG-16. There are a few pooling techniques, here we will see two of those techniques.

2.8.3.1. Average Pooling

Average pooling averages the values within the window per channel. The Sub-indices of a window $p_{(I,J)}^{(k-1)}$are defined as

$$p_{(I,J),i,a,b}^{(k-1)} \in R$$

Using this definition, average pooling can be defined as

$$\psi_{(k)}^{(maxpool)}\left(o^{(k-1)}\right) = o^{(k)} = \{o_{(I,J),i}^{(k)} | \forall((I,J),i)(\exists p_{(I,J),i}^{(k-1)})[o_{(I,J),i}^{(k)} = \sum_{a=1}^{K}\sum_{b=1}^{K}\frac{p_{(I,J),i,a,b}^{(k-1)}}{K^2}]\}$$

In other words, for every index $(I,J),i$, there exists a K x K matrix. The value of the output at index $(I,J),i$ is defined as the average value of that matrix.

2.8.3.2. Max Pooling

As it can be depicted from its name, max pooling takes the largest value in a certain channel within a window. Let's define the first sub-index of a window as if it is referring to a node as

$$p_{(I,J),i}^{(k-1)} \in \mathrm{R}^{KxK}, 0 < i < m^{(k-1)}$$

Using this definition, max pooling can be defined as

$$\psi_{(k)}^{(maxpool)}\left(o^{(k-1)}\right) = o^{(k)} = \{o_{(I,J),i}^{(k)} | \forall((I,J),i)(\exists p_{(I,J),i}^{(k-1)})[o_{(I,J),i}^{(k)} = \max(p_{(I,J),i}^{(k-1)})]\}$$

In other words, for every index $(I,J),i,$ there exists a K x K matrix. The value of the output at index $(I,J),i$ is defined as the maximum value for the matrix. This is illustrated in figure1 below. Max pooling is mostly used after the first or second convolutional layer to reduce the dimensionality of the input in classification tasks.

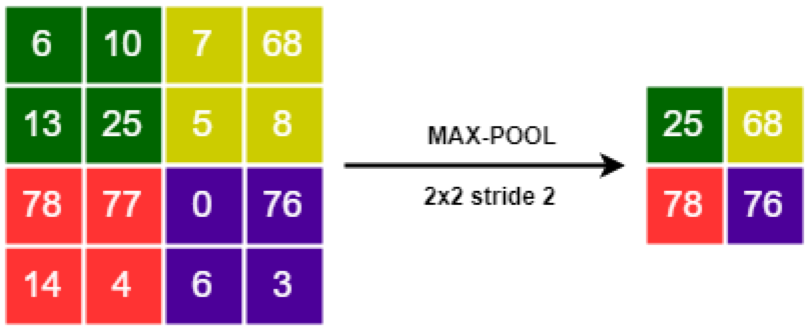


Figure 12. Maxpool operation

2.8.4. Local Response Normalization (LRN)

Local response normalization layers are designed to normalize the response of neurons in the same spatial location, but from different feature maps; normalization is across the depth. This is loosely equivalent to averaging a neuron’s output with those of other neurons at the same location in different feature maps. The computation can be formulated as follows:

$$b_{x,y}^{i} = \frac{a_{x,y}^{i}}{(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a_{x,y}^{j})^2)^\beta}$$

The terms $a_{x,y}^{i}$ and $b_{x,y}^{i}$ are input and output neurons at location <x , y> of feature map i, respectively. N refers to the total number of feature maps in the layer and n refers to the number of adjacent feature maps to be considered. The constants k, n, α, and β are non-trainable parameters, also known as hyper-parameters. Their values are selected from validation where the best set of values is chosen based on many trial runs.

It is worth noting that the ordering of feature maps is arbitrary and it is up to the network itself to adjust the feature extractors to adapt to the normalizations.

As it pointed out in, this response normalization method bares some similarity to “lateral inhibition” behaviour found in biological neurons. Now what does “lateral inhibition” mean? This refers to the capacity of an excited neuron to subdue its neighbours. We basically want a significant peak so that we have a form of local maxima. This tends to create a contrast in that area, hence increasing the sensory perception. Increasing the sensory perception is a good thing! We want to have the same thing in our CNNs.

Local Response Normalization (LRN) layer implements the lateral inhibition we were talking about in the previous section. This layer is useful when we are dealing with ReLU neurons. Why is that? Because ReLU neurons have unbounded activations and we need LRN to normalize that. We want to detect high frequency features with a large response. If we normalize around the local neighbourhood of the excited neuron, it becomes even more sensitive as compared to its neighbours.

At the same time, it will dampen the responses that are uniformly large in any given local neighbourhood. If all the values are large, then normalizing those values will diminish all of them. So basically we want to encourage some kind of inhibition and boost the neurons with relatively larger activations.

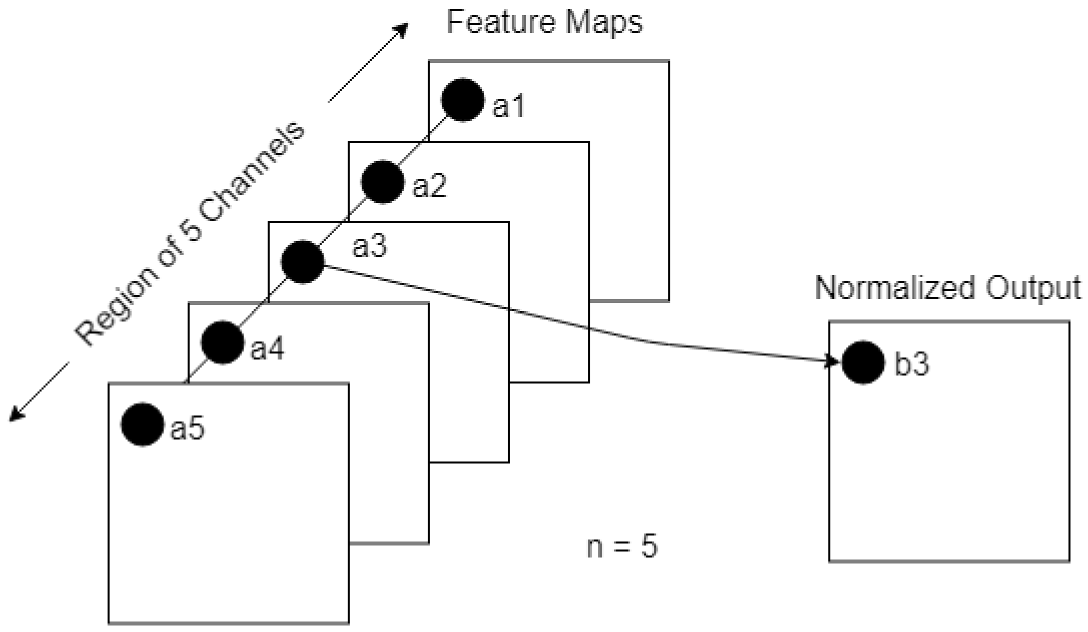


Figure 13. Normalization through channels

2.8.5. Fully Connected Layers

Neurons between two adjacent layers are fully pair wise connected, while in neurons from the same layer are not. It Computes the class probability scores by outputting a vector of C dimensions, with C being the number of classes. All neurons are connected to this layer. They can be interpreted as 1x1 convolutional layers.

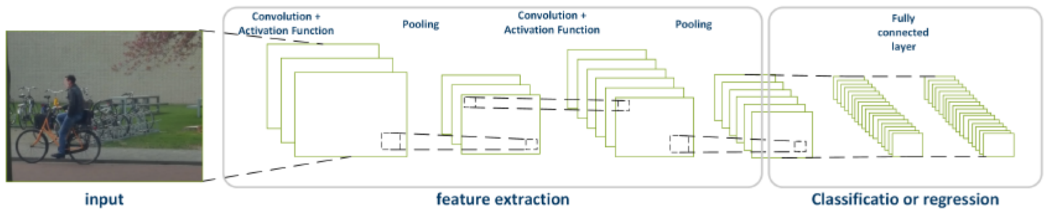

Convolution + Activation Function
Pooling
Convolution + Activation Function
Pooling
Fully connected layer
input
feature extraction
Classificatio or regression

Figure 14. Fully Connected Layer

Chapter 3 : Software Aspects

3.1. Frameworks

For ease of DNN development and to enable sharing of trained networks, several deep learning frameworks have been developed from various sources. These open source libraries contain software libraries for DNNs. Caffe was made available in 2014 from the University of California Berkeley (UC Berkeley) . It supports C, C++, Python, and MATLAB. Tensorflow was released by Google in 2015, and supports C++ and python; it also supports multiple CPUs and GPUs and has more flexibility than Caffe, with the computation expressed as dataflow graphs to manage the tensors (multidimensional arrays). Another popular framework is Torch, which was developed by Facebook and NYU and supports C, C++ and Lua; PyTorch is its successor and is built in Python. There are several other frameworks such as Theano, MXNet, CNTK. There are also higher level libraries that can run on top of the aforementioned frameworks to provide a more universal experience and faster development. One example of such libraries is Keras, which is written in Python and supports Tensorflow, CNTK, and Theano.

3.2. Models

Pretrained DNN models can be downloaded from various websites for the various different frameworks. It should be noted that even for the same DNN (e.g., AlexNet) the accuracy of these models can vary by around 1%–2% depending on how the model was trained, and thus the results do not always exactly match the original publication. The source code of the used python code is attached in the appendix section.

3.3. Datasets

In this section, we will focus on some datasets that have been used in training and validation of each neural network. Since we are mostly focusing on convolutional neural networks, we will look at 3 image classification datasets.

3.3.1. MNIST Dataset

MNIST dataset consists of 60.000 training samples and 10.000 test samples. Each sample is a 28x28 black and white image of a handwritten digit (0 to 9). To our knowledge, the best model trained on MNIST achieved almost zero (0:23%) error rate.

3.3.2. CIFAR10 dataset

The CIFAR-10 dataset consists of 60000 32x32 coloured images from 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. To our knowledge, the best model trained on CIFAR10 achieved (3:47%) error rate.

3.3.3. ImageNet Dataset

The dataset used in ILSVRC (Imagenet Large Scale Visual Recognition Challenge) is called ImageNet. ImageNet comes with 1281167 training images and 50000 validation images consisting of 1000 classes containing multiple dog species and daily objects. ImageNet comes with bounding boxes showing where the object is in the image. We are not interested in the object detection task, but we focus on th classification of the object after it has been detected. The best submission from 2016 challenge has achieved 0.02991 error rate. This is equal to 97:009% top-5 accuracy. In our design, we will be building our design for the classification of this dataset.

3.4. AlexNet

AlexNet was designed and developed by Alex Krizhevsky n 2012 for the ImageNet Large Scale Visual Recognition Competition (ILSVRC). AlexNet is a deep convolutional neural network (CNN) which was used to classify the 1.2 million images into 1000 classes in the ImageNet challenge. AlexNet was able to achieve considerably better results than previous models such as LeNet. The difference between AlexNet and previous models is the number layers and parameters. AlexNet consists of five CONV layers. Some of those layers are followed by max-pooling layers. There are also three fully connected layers, and a final 1000-way softmax that picks between the Imagenet classes.

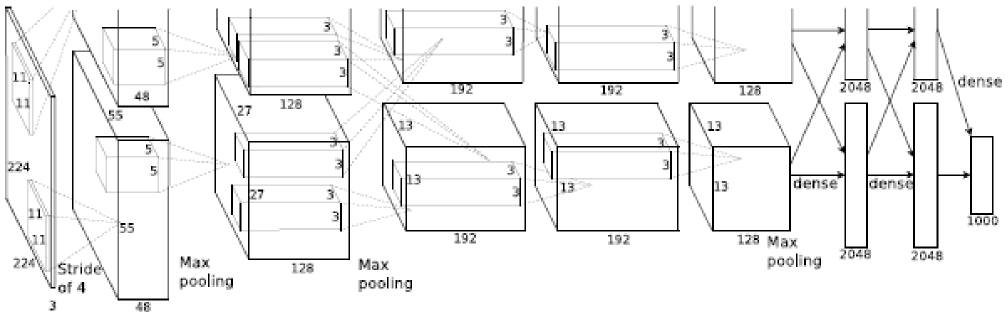


Figure 15. AlexNet Architecture

AlexNet was the first CNN to win the ImageNet Challenge in 2012. It consists of five CONV layers and three FC layers. Within each CONV layer, there are 96 to 384 filters and the filter size ranges from 3 × 3 to 11 × 11, with 3 to 256 channels each. In the first layer, the three channels of the filter correspond to the red, green, and blue components of the input image (RGB). A ReLU nonlinear activation function is used in each layer. Max pooling of 3×3 is applied to the outputs of layers 1, 2, and 5. To reduce computation, a stride of 4 is used at the first layer of the network. AlexNet introduced the use of Local Response Normalization (LRN) in layers 1 and 2 before the max pooling, though LRN is no longer popular in later CNN models. One important factor that differentiates AlexNet from LeNet is that the number of weights is much larger and the shapes vary from layer to layer. To reduce the amount of weights and computation in the second CONV layer, the 96 output channels of the first layer are split into two groups of 48 input channels for the second layer, such that the filters in the second layer only have 48 channels. Similarly, the weights in fourth and fifth layers are also split into two groups. In total, AlexNet requires 61 million weights and 724 million MACs to process one 227 × 227 input image.

3.5. FPGAs versus GPUs

A field-programmable gate array (FPGA) is an integrated circuit that contains an array of programmable logic blocks (gates/LUTs) and a hierarchy of reconfigurable interconnects. It also contains memory elements called flip-flops or more complex memory blocks such as Block RAMs. It contains some powerful computation circuits called DSPs as well. It may contain other blocks such as - MACs, Oscillators, and PLLs. The FPGA configuration is generally specified using a hardware description language (HDL). One of the main advantages of the modern day FPGAs is large diversity of interface or in Other Words an FPGA is a “flexible” hardware that allows its user to configure it according to their considerations and constraints in an almost optimal way. It allows them to manage somewhat limited resources and obtain the desired specifications such as power consumption, area, and speed. Unlike GPU that multi-threads and consume resources (overhead) and sometimes is not “real”, in FPGA parallelism is very natural.

Over the past several years, Graphics Processing Units (GPUs) have become the standard for implementing deep learning algorithms in computer vision and other applications. GPUs offer a large number of processing elements, a stable and expanding ecosystem, support for standards such as OpenCL, and a wide range of intellectual property to develop applications rapidly. However, as the industry matures, field programmable gate arrays (FPGAs) are now starting to emerge as credible competition to GPUs for implementing deep learning algorithms. A recently published paper from Microsoft Research garnered quite a bit of attention in the industry when it contended that using FPGAs could be as much as 10 times more power efficient compared to GPUs. Although the performance of FPGAs was much lower than GPUs, the FPGA used for comparison was a mid-range device, which left the door open for further lowering the power on FPGAs. The fact that power consumption was much lower could have significant implications on many applications where high performance may not be the top priority.

FPGAs, in essence, make hardware “soft”. It is possible to reconfigure FPGAs on the fly, making them as adaptable as GPU software. FPGA manufacturers take pride in the fact that they have an advantage over GPUs in terms of performance per watt, and have begun to make further investments in deep learning as a target market. Intel’s Altera group, for example, has published a white paper that talks about using OpenCL for FPGA development. Meanwhile, Xilinx recently made an investment in a startup called Teradeep that accelerates deep learning algorithms on a Xilinx FPGA platform. While FPGAs may look attractive based on the power reduction, there are few areas where they need to gain ground on GPUs before they can become a viable alternative. The development flow, for instance, is radically different for FPGAs compared to GPUs. The GPU runs software, while the FPGA runs hardware. One can easily take a software program and run it on a GPU. For FPGAs, however, one has to convert the software algorithm into hardware blocks before it can be mapped onto the FPGA. Given the complex nature of deep learning algorithms, mapping them to hardware becomes an extremely complicated problem and there are many possible solutions that are currently only being studied in academia. A recent UCLA paper states that optimal implementation of convolutional neural networks (CNNs) on an FPGA platform is a function of computational resources and represents a trade-off in terms of memory bandwidth. If the hardware is not carefully designed, its computing throughput may not match the memory bandwidth, leading to a significant degradation of performance. Given these issues, converting algorithms to hardware remains a challenge for FPGA implementation. Fig.() shows a comparison between the FPGAs and GPUs

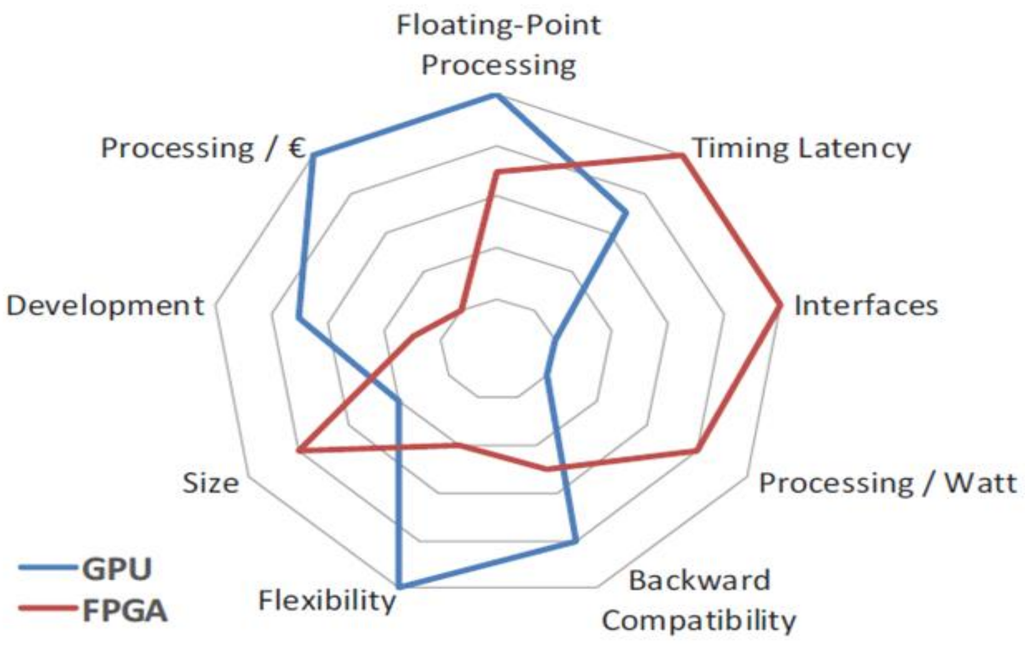
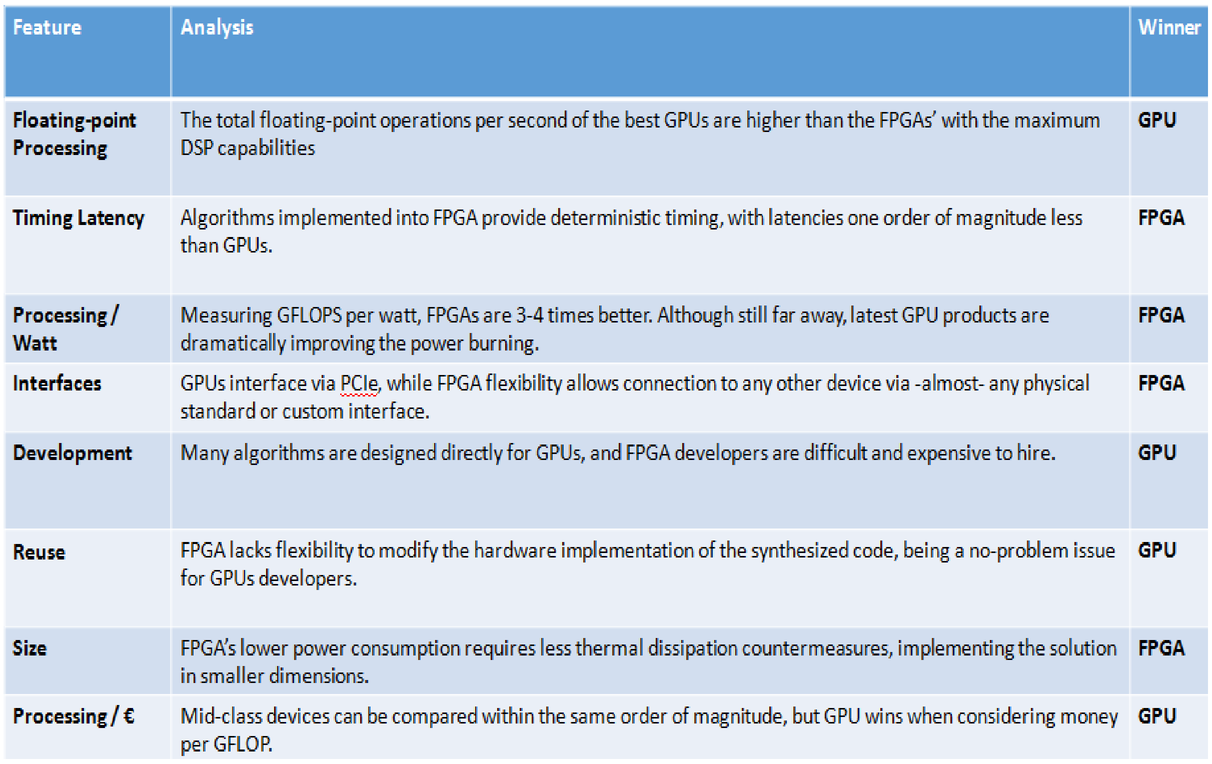


Figure 16. FPGA vs GPU

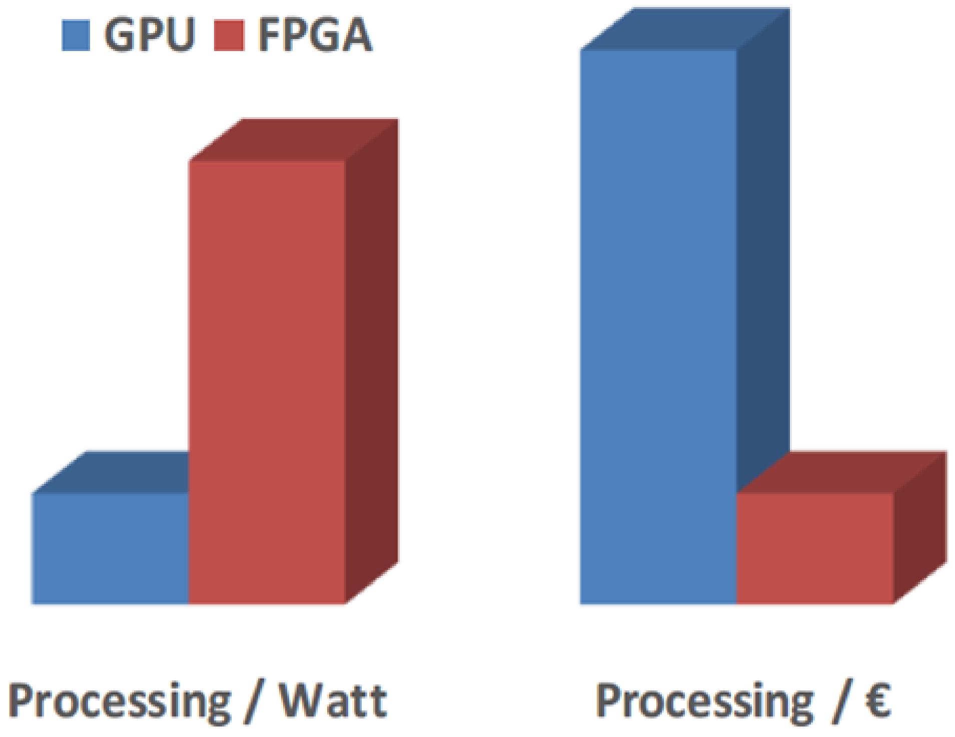
The following table illustrates the previous fig.() showing exactly which platform is better



Feature	Analysis	Winner
Floating-point Processing	The total floating-point operations per second of the best GPUs are higher than the FPGAs' with the maximum DSP capabilities	**GPU**
Timing Latency	Algorithms implemented into FPGA provide deterministic timing, with latencies one order of magnitude less than GPUs.	**FPGA**
Processing / Watt	Measuring GFLOPS per watt, FPGAs are 3-4 times better. Although still far away, latest GPU products are dramatically improving the power burning.	**FPGA**
Interfaces	GPUs interface via PCIe, while FPGA flexibility allows connection to any other device via -almost- any physical standard or custom interface.	**FPGA**
Development	Many algorithms are designed directly for GPUs, and FPGA developers are difficult and expensive to hire.	**GPU**
Reuse	FPGA lacks flexibility to modify the hardware implementation of the synthesized code, being a no-problem issue for GPUs developers.	**GPU**
Size	FPGA's lower power consumption requires less thermal dissipation countermeasures, implementing the solution in smaller dimensions.	**FPGA**
Processing / €	Mid-class devices can be compared within the same order of magnitude, but GPU wins when considering money per GFLOP.	**GPU**

FPGAs also are not good at training for deep learning algorithms. FPGAs have a limited number of equivalent gates in each product family, capping the number of nodes one can use for training. Training is also much more computationally intensive, requiring constant adjustment of parameters and making it harder to implement the algorithm architecturally in hardware. In practical terms, this may not be as much of a concern for FPGA vendors, as deployed CNNs are used in the final device and that is where the volume comes from.

The overall ecosystem for deep learning application development for FPGAs is also in its early stages and there are not currently enough vendors who offer solutions to accelerate time to market for deep learning applications. Also, training engineers on both deep learning and hardware is another issue that the FPGA vendors will need to address. As it is, there is a shortage of engineers who understand deep learning. The number of engineers who have an understanding of deep learning in addition to the hardware development process is even smaller. A good digital signal processor (DSP) hardware engineer may be trained to learn about computer vision algorithms, but training them to become expert at CNNs would be a greater challenge. Given this, FPGAs could be a platform of choice for well-understood deep learning problems. In the long run, if and when FPGA vendors overcome some of these challenges, they will become more formidable competitors to GPU manufacturers in the deep learning market. But who knows, by then the GPU manufacturers may have invented something new to keep them ahead. Fig.() shows how FPGA excels in the resources optimization when it comes to power and cost.

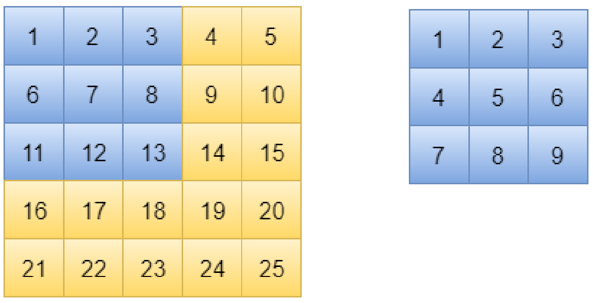


Chapter 4 : System Architecture

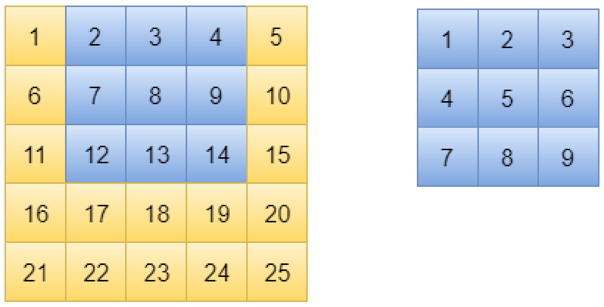
4.1. Convolution operation

If two inputs are convolved together, one of them slides over the other and each iteration the output of the multiplication of each corresponding elements is computed. This is the normal convolution process.

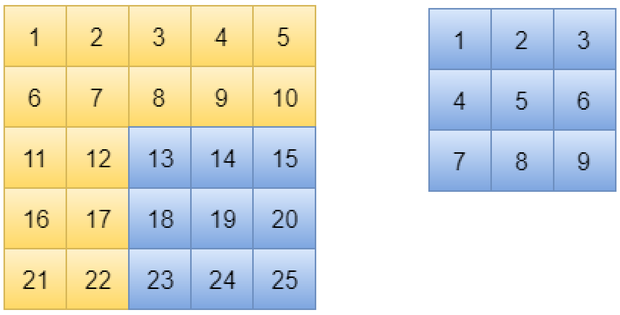
An example to illustrate the concept behind the convolution; let's assume we have two matrices that we want to get the output matrix of their convolution. One of them is 3x3 and the other is 5x5. Initially, the 3x3 matrix resides the first corner part of the 5x5 matrix as shown in fig.().



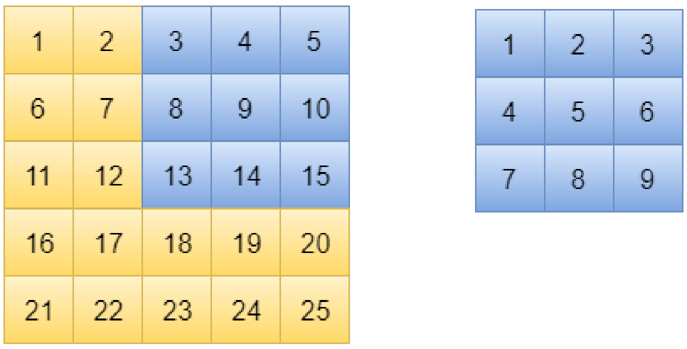
We are going to multiply all these values together to get the first value of the output matrix. Then, we are going to slide the 3x3 filter one position to the right as shown in fig.()



to get the next value of the output, we continue like this till we get the final value in the bottom right corner of the output matrix as depicted in fig.(). The output will be a 3x3 matrix.



Now, let’s examine the effect of the stride on the output. The stride of the convolution determines the sliding jump. Fig.() shows the sliding of the window to the right one time, this is different from the first example as the output will be 2x2 in this case.



Some strides may not fit well with some matrices, For example, a 3x3 window will not fit well with a 5x5 input matrix when the stride is 3. This is depicted in fig.() as after the first slide, the second one can’t be performed as the window will get out of bonds of the input matrix. In some cases zero padding is used to accommodate for these problems.

Generally, when two matrices undergo convolution, the output matrix size can be calculated as follows:

$$M = \frac{(N - F)}{S} + 1$$

Where **M**, **N**, **F** and **S** are the output matrix size, input matrix size, window size and stride respectively.

So using the previous equation, we get the final output matrix as shown in the following table:

stride	Equation	Final size
1	(5-3)/1 + 1	3x3
2	(5-3)/2 + 1	2x2
3	(5-3)/3 + 1	1x1

4.2. Network Architecture

The main building block in our design is the Processing Element (PE). At any point of time, a PE is equivalent to a neuron. The PE consists of two Register Files (RF) responsible for storing some pixels of the input as well as the current filter, a multiplier circuit to multiply those values, an adder and an accumulator to compute the partial sums. Each operation in the PE is labelled Multiply and Accumulate (MAC). The design of the PE is illustrated in fig.().

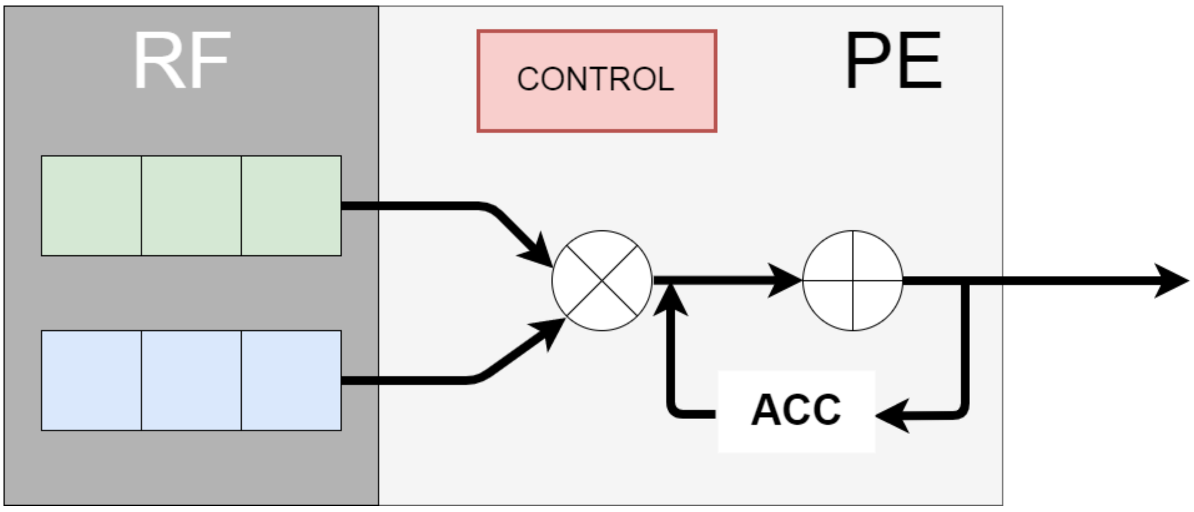


Figure 17. Processing Element

The main computational power in this design lies upon a matrix of PEs shown in fig.(). One of the main features of the design is using an array of 168 PEs that are mapped as a matrix of 14x12 PEs to compute the output of all layers. The reasons behind the choice of this configuration will be discussed later on.

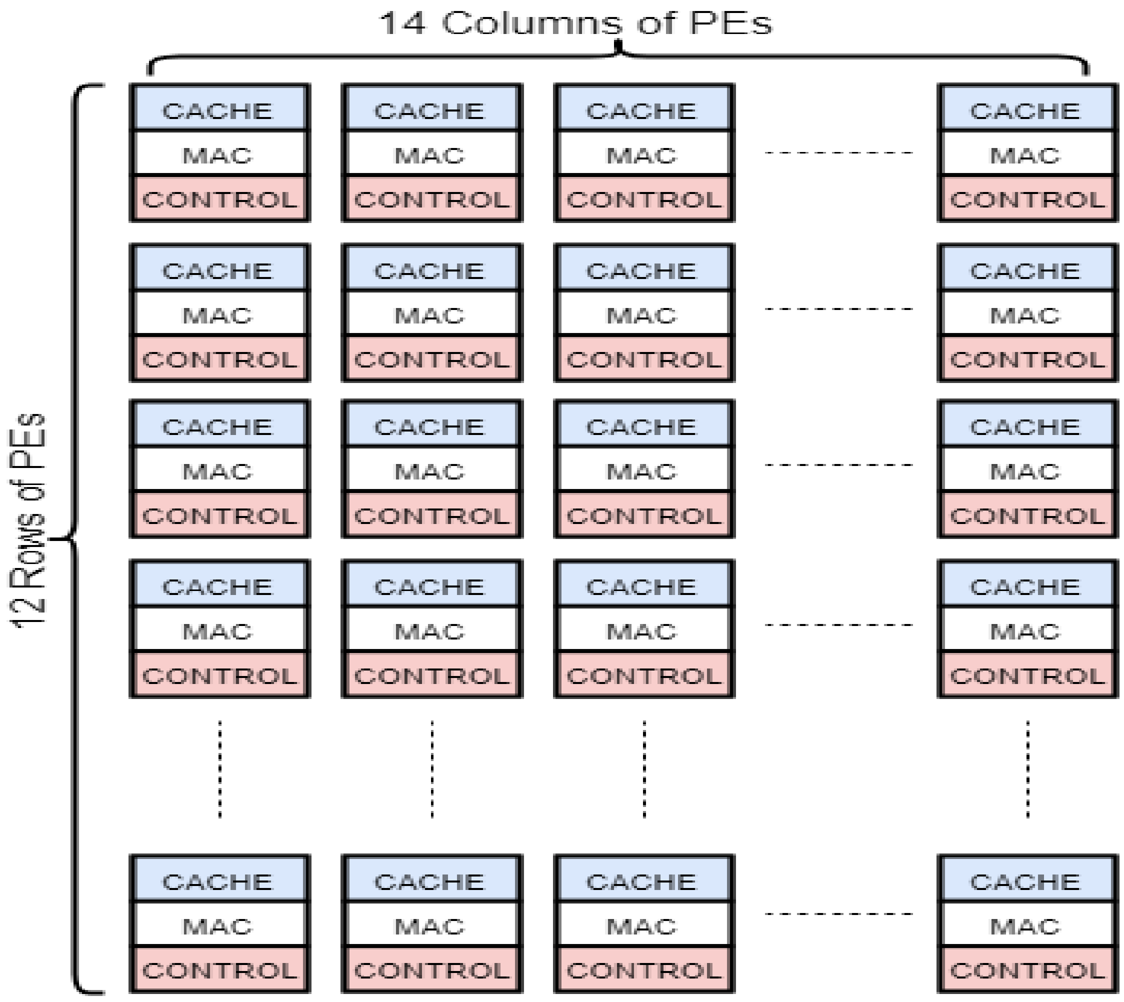


Figure 18. Matrix of PEs

Due to the massive size of the filter weights, it’s not applicable to store all of them at once due to hardware and memory limitations; instead some buffers are used to store a portion of these weights, as well as the input needed for convolution, and the generated partial sums. Four buffers are used which are labelled the Filter, PSUM, Swapping 1 and Swapping 2 buffers. The importance of each buffer will be discussed in a following subsection.

4.3. Dataflow

Now, we will focus on how various data (input pixels, weights, biases and output pixels) is stored in this matrix. The technique used in the dataflow is row-stationary. A row stationary dataflow aims to maximize the reuse and accumulation at the Register

File (RF) level for all types of data (here, we focused on pixels and partial sums) for the overall energy efficiency. This differs from Weight Stationary or Output Stationary dataflow techniques, which optimize for only weights and partial sums, respectively. The row stationary dataflow assigns the processing of a 1-D row convolution into each PE for processing as shown in Fig.().

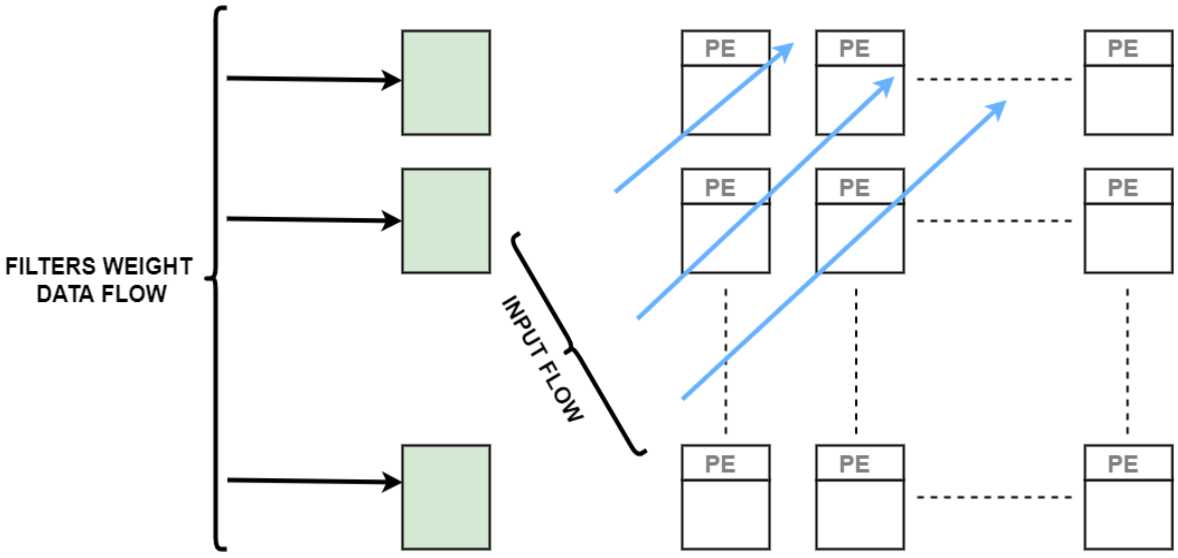


Figure 19. Dataflow into the PEs

It keeps the row of filter weights, or multiple rows of one filter or multiple filters depending on each layer requirements, stationary inside the RF of the Processing Element (PE) and then streams the input pixels into the PE. The PE does the Multiply and Accumulate (MACs) for each sliding window at a time, which uses just one memory space for the accumulation of partial sums. Since there are overlaps of input pixels between different sliding windows, the input pixels can then be kept in the RF and get reused. By going through all the sliding windows in the row, it completes the 1-D convolution and maximizes the data reuse and local accumulation of data in this row. With each PE processing a 1-D convolution, multiple PEs can be aggregated to complete the 2-D convolution as shown in Fig.(). For example, to generate the first row of output pixels with a filter having three rows, three 1-D convolutions are required. Therefore, we can use three PEs in a column, each running one of the three 1-D convolutions. The partial sums are further accumulated vertically across the column of PEs to generate the first output row. To generate the second row of output, we use another column of PEs, and use the same rows of filters to perform the 1-D convolutions. Additional columns of PEs are added until all rows of the output are completed (i.e., in most cases the number of PE columns equals the number of output rows). This 2-D array of PEs enables other forms of reuse to reduce accesses to the more expensive global buffer. For example, each filter row is reused across multiple PEs horizontally. Each row of input pixels is reused across multiple PEs diagonally, and each row of partial sums is further accumulated across the PEs vertically. Therefore, 2-D convolutional data reuse and accumulation are maximized inside the 2-D PE array.

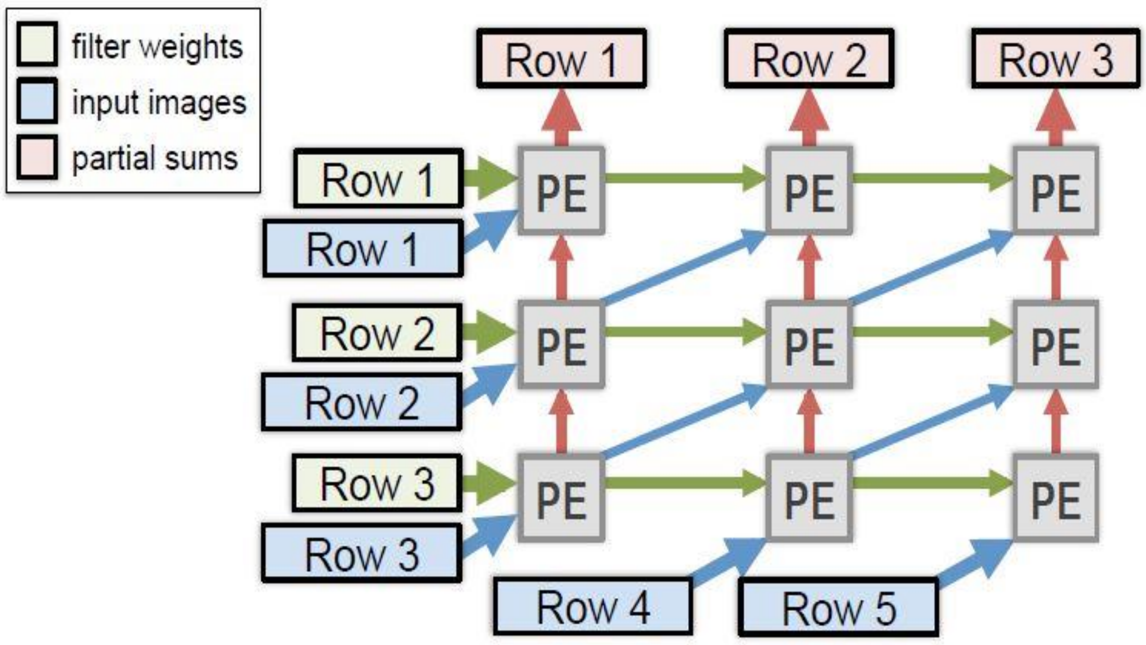


Figure 20. Generation of each Row

To address the high-dimensional convolution of the CONV layer (i.e., multiple fmaps, filters, and channels), multiple rows can be mapped onto the same PE as shown in Fig.(). The 2-D convolution is mapped to a set of PEs, and the additional dimensions are handled by interleaving or concatenating the additional data. For filter reuse within the PE, different rows of fmaps are concatenated and run through the same PE as a 1-D convolution. For input fmap reuse within the PE, different filter rows are interleaved and run through the same PE as a 1-D convolution. Finally, to increase local partial sum accumulation within the PE, filter rows and fmap rows from different channels are interleaved, and run through the same PE as a 1-D convolution. The partial sums from different channels then naturally get accumulated inside the PE.

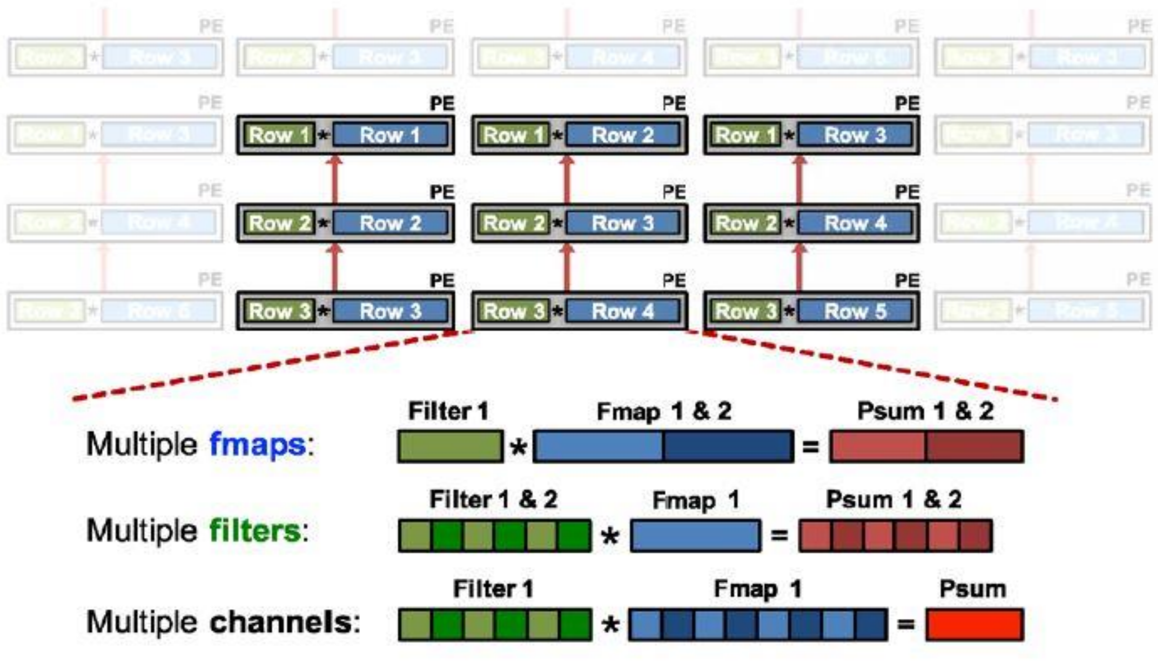
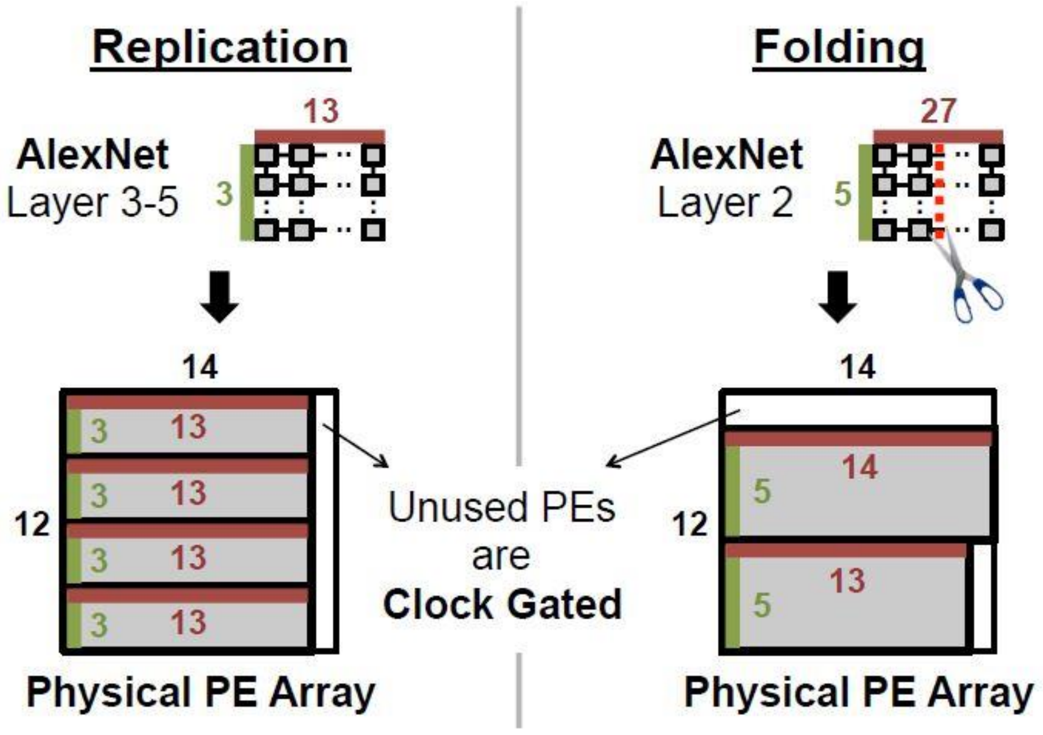


Figure 21. Row Stationary

The number of filters, channels, and fmaps that can be processed at the same time is programmable, and there exists an optimal mapping for the best energy efficiency, which depends on the shape configuration of the DNN as well as the hardware resources provided, e.g., the number of PEs and the size of the memory in the hierarchy. Since all of the variables are known before runtime, it is possible to build a compiler (i.e., mapper) to perform this optimization offline to configure the hardware for different mappings of the RS dataflow for different DNNs.

In order to support the RS dataflow, there exists a problem that needs to be solved in the hardware design. How can the fixed-size PE array accommodate different layer shapes?

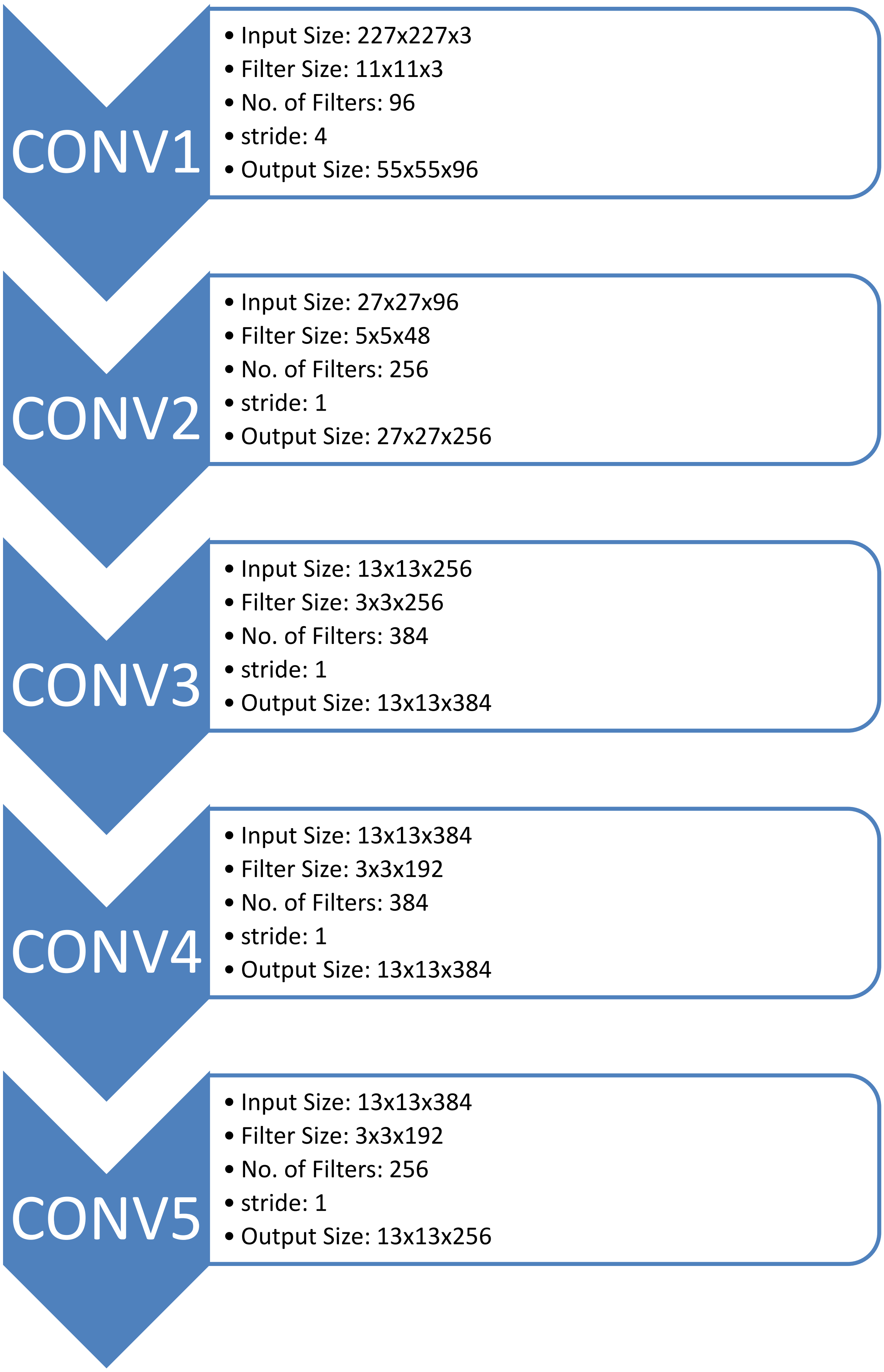
Two mapping strategies can be used to solve this problem as shown in Fig.(). First, replication can be used to map shapes that do not use up the entire PE array. For example, in the third to fifth layers of AlexNet, each 2-D convolution only uses a 13×3 PE array. This structure is then replicated four times, and runs different channels and/or filters in each replication. The second strategy is called folding. For example, in the second layer of AlexNet, it requires a 27×5 PE array to complete the 2-D convolution. In order to fit it into the 14×12 physical PE array, it could be folded into two parts, 14×5 and 13×5, and each is vertically mapped into the physical PE array. However, replication is favorable in this scenario.



Since not all PEs are used by the mapping, the unused PEs can be clock gated to save energy consumption. The simplest way to pass data to multiple destinations is to broadcast the data to all PEs and let each PE decide if it has to process the data or not.

4.4. Convolutional Layers

The AlexNet architecture implemented in the software algorithm consists of five convolutional layers:



CONV1

- Input Size: 227x227x3
- Filter Size: 11x11x3
- No. of Filters: 96
- stride: 4
- Output Size: 55x55x96

CONV2

- Input Size: 27x27x96
- Filter Size: 5x5x48
- No. of Filters: 256
- stride: 1
- Output Size: 27x27x256

CONV3

- Input Size: 13x13x256
- Filter Size: 3x3x256
- No. of Filters: 384
- stride: 1
- Output Size: 13x13x384

CONV4

- Input Size: 13x13x384
- Filter Size: 3x3x192
- No. of Filters: 384
- stride: 1
- Output Size: 13x13x384

CONV5

- Input Size: 13x13x384
- Filter Size: 3x3x192
- No. of Filters: 256
- stride: 1
- Output Size: 13x13x256

4.4.1. First Convolutional Layer

The input of this layer is the input image to the network, so its size is 227x227x3. This layer consists of 96 filters (kernels) each having a size of 11x11x3. The output of this layer is 55x55x96. As discussed in the row stationary technique, each PE should store one row of the filter as well as one row of the input feature map. Moreover, the accumulation of each row of the output feature map is done vertically. So, to compute one row of the output, 11 PEs should be stacked together vertically as there are 11 rows in each filter.

Due to the massive size of the input and the FPGA memory limitations, the input is divided into 4 parts; each part consists of 63 rows and is stored one at a time in one of the buffers called swapping 1 buffer. This will produce 14 rows of the output feature maps when convolved with the filters. It was chosen that the filters are divided into 6 groups of 16 filters each due to the same problem. Those filters will be stored with their corresponding biases in the filter buffer. So, the matrix of the PEs should be 11x14 regarding layer 1.

The stride in this layer is 4. So after each 4 rows of the input are stored, the top PE in the next column is activated for storage to emphasize the impact of the vertical stride. For example, when storing the first row of the input, only one column is activated. After 4 rows, when starting to store the fifth row, the first PE in the second column is activated whilst the first column is still active, meaning that the fifth row will be written in the fifth PE of the first column as well as the first PE in the second column. After another 4 rows, when starting to store the ninth row, the first PE in the third column is activated whilst the first and second columns are still active, meaning that the ninth row will be written in the ninth PE of the first column, as well as the fifth PE of the second column and the first PE in the third column. The storage of the filter is a straight forward task, as each row of PEs stores the same row of the filter. Note that only one depth of the filter and input is stored in the PE each time.

Now, we'll focus on how the convolution operation is performed. As mentioned before, to produce an output pixel, the summations of the multiplications of all the corresponding filter weights and input pixels on all depths have to be computed. This is computed in multiple iterations, as one depth only is present in the PE each time. Each PE holds 16 rows of the same depth from 16 different filters in one of the RFs and 1 row of the input of that depth in the other RF. The outputs of the convolution of each filter row and input row are summed vertically to produce the partial sums (PSUMs) of one row of one of the output feature maps. Here, the depth is 3 so we have to store these partial sums to be able to add them to the other partial sums produced from other depths of the same filter and the same part of the feature map. Hence, those partial sums are stored in the PSUM buffer. The size of this buffer from the point of view of the first layer should be 14x55x16. Where 14 represents the number of rows of each output feature map, 55 represents the size of each row, and 16 represents that the number of filters produced each iteration.

It is noted that the 14 pixels produced are from different rows. Therefore, when storing the output PSUMs, it was taken into consideration that each pixel is written 55 places apart from the next and preceding ones, where 55 indicates the size of the whole row which is the exact distance between every two pixels. After the convolution of the first filter has finished, the RF of the filters is adjusted to start outputting the second filter and then the third filter and so on. This process is repeated 16 times as the number of filters is 16. After the end of this process, the partial sums produced from one depth of the 16 filters are now stored in the PSUM buffer. The second depth of the input and the filters is then stored in the RFs and the process is repeated. Note that the output PSUMs in this stage aren't stored directly in the PSUM buffer, rather they are added to their corresponding PSUMs from the first depth, and the result of both of them is stored in the PSUM buffer as shown in fig.(). Finally, the third depth is done and added to the previous two, after that the bias is added to the final output. Note that each filter has one bias.

	Storage from the point of view of layer 1
Filter Buffer	11x11x3x16 + 16 = 5824 locations
PSUM Buffer	14x55x16 = 12320 locations
Swapping Buffer 1	227x63x3 = 42903 locations
Swapping Buffer 2	------

Now, the 14 rows of the 16 feature maps are present in the PSUM buffer, the maxpooling layer takes them from the PSUM buffer and does the maxpooling on those

PSUMs then store the new values in the swapping buffer 2 so that the PSUM buffer is ready for the next iteration of either filters or inputs. The maxpool operation is discussed in the next section. The previous process is repeated for the other 5 groups of the filters. After, that this part of the input is no longer needed, so the next part of the input is stored in the swapping buffer 1 from the DRAM and the process is repeated, then the third part and finally the last part. Note that the last part of the input contains 59 rows only as only 13 rows remain in each output feature map.

	Starting Row	Last Row	Number of Rows	Output FMaps Rows
First part	1	63	63	14
Second part	57	119	63	14
Third part	113	175	63	14
Fourth part	169	227	59	13

4.4.2. Second Convolutional Layer

The input of this layer is the output of the first Local Response Normalization Layer, so its size is 27x27x96. It is stored in swapping buffer 2. This layer consists of 256 filters (kernels) each having a size of 5x5x48. The new feature added to this layer is that the filters are divided into two groups; the first groups consists of the first 128 filters and they get convolved with the first half of the number of depths of the input (i.e. first 27x27x48), the second group consists of the remaining 128 filters and they get convolved with the other half of the number of depths of the input (i.e. last 27x27x48). The output of this layer is 27x27x256. The zero padding technique is used here to ensure that the output feature maps spatial dimensions are the same as those of the input feature maps (i.e. 27x27). As discussed in the row stationary technique, each PE should store one row of the filter as well as one row of the input feature map. Moreover, the accumulation of each row of the output feature map is done vertically. So, to compute one row of the output, 5 PEs should be stacked together vertically as there are 5 rows in each filter.

Due to the presence of the two groups, the input is divided into 2 parts; each part consists of 48 depths. The first part will produce the first 128 output feature maps when convolved with the first 128 filters, and so the second part will produce the remaining

128 feature maps. It was chosen that the filters of each part are divided into 9 batches of 15 filters each with the exception of the ninth batch which contains 8 filters only. Those filters will be stored with their corresponding biases in the filter buffer. So, the matrix of the PEs should be 5x27 regarding layer 2. Due to the restrictions imposed by layer 1, the number of rows should be at least 11. So, to accelerate the design, the matrix of the PEs in layer 2 will be considered as 10x14 where each output pixel will not the output of one depth only but rather two depths from the same filter.

The stride in this layer is 1. So, after each 1 row of the input is stored, the top PE in the next column is activated for storage to apply the vertical stride. For example, when storing the first row of the input, only one column is activated. When starting to store the second row, the first PE in the second column is activated whilst the first column is still active, meaning that the second row will be written in the second PE of the first column as well as the first PE in the second column. When starting to store the third row, the first PE in the third column is activated whilst the first and second columns are still active, meaning that the third row will be written in the third PE of the first column, as well as the second PE of the second column and the first PE in the third column. This means that the input rows are placed diagonally in the PEs. When the first 5 rows are stored, no other PEs are activated in the same column at this time, as they will be used for the storage of another depth to keep the vertical summation valid, meaning no extra hardware has to be added to that used in layer 1. Note that in this layer, 8 depths of the filter and input are stored in the PE each time. After the whole depth of the input is stored, another depth is then stored in the same five rows of PEs (14x5), then the third depth is stored again till 8 depths are stored, after that the second part of PEs (14x5) are activated and the same operation is repeated for the lower half of PEs. The storage of the filter is a straight forward task, as each row of PEs stores the same row of the filter. So, each PE from the first half will contain the first 8 depths of one filter and the second half will contain the second 8 depths of the same filters.

	Starting Row	Last Row	Number of depths	Output FMaps Rows
First part – First 48 depths	1	16	48	14
Second part – First 48 depths	13	27	48	13
First part – Second 48 depths	1	16	48	14
Second part – Second 48 depths	13	27	48	13

Now, we'll focus on how the convolution operation is performed. As mentioned before, to produce an output pixel, the summations of the multiplications of all the corresponding filter weights and input pixels on all depths have to be computed. This is computed in multiple iterations, as 8 depths only are present in the PE each time. Each PE holds 1 row of the same filter from 8 different depths in one of the RFs and 1 row of the input corresponding to each of those 8 depths in the other RF. The outputs of the convolution of each filter row and input row are summed vertically to produce the partial sums (PSUMs) of one row of one of the output feature maps due to two depths. This operation is repeated 8 times to produce the partial sums (PSUMs) of one row of one of the output feature maps due to 16 depths. Only 14 columns of PEs are present so the output will be 14 rows. Here, the depth of each filter in each group is 48 so we still have to store these partial sums in the PSUM buffer. After the operation is completed, another filter with 16 depths is stored in the matrix. This operation is repeated till all the 15 filters are passed on the first part of the input. After that, the rest of the input is brought to the PEs with the same 16 depths, and the first filter with the same 16 depths is brought again. The process is repeated till the 15 filters are passed on the remaining part of the input.

At this point, the PSUM now contains the partial sums contain the whole 15 output feature maps, but due to 16 depths only. Now the whole process is repeated using another 16 depths from the input and the filters, then, the output PSUMs in this stage aren’t stored directly in the PSUM buffer, rather they are added to their corresponding PSUMs from the first 16 depths, and the result is stored in the PSUM buffer. This will be repeated a final time to compute the output of the convolution due to the remaining 16 depths. The bias is then added to the output PSUM. Note that between the finish of the 15 filters in a batch and the storage of another 15, the maxpooling layer operates to store the outputs in the swapping buffer 1 again. The maxpooling layer will be discussed in the next section.

The size of the PSUM buffer from the point of view of the second layer should be 27x27x15. Where 27 represents the number of rows of each output feature map, 27 represents the size of each row, and 15 represents that the number of filters produced each iteration.

	Storage from the point of view of layer 2
Filter Buffer	5x5x48x15 + 15 = 18015 locations
PSUM Buffer	27x27x15 = 10935 locations
Swapping Buffer 1	27x27x96 = 69984 locations
Swapping Buffer 2	------

4.4.3. Third Convolutional Layer

The input of this layer is the output of the second Local Response Normalization Layer, so its size is 13x13x256. It is stored in swapping buffer 2. This layer consists of 384 filters (kernels) each having a size of 3x3x256. The output of this layer is 13x13x384. The zero padding technique is used here to ensure that the output feature maps spatial dimensions are the same as those of the input feature maps (i.e. 13x13). As discussed in the row stationary technique, each PE should store one row of the filter as well as one row of the input feature map. Moreover, the accumulation of each row of the output feature map is done vertically. So, to compute one row of the output, 3 PEs should be stacked together vertically as there are 3 rows in each filter.

The input is divided into 4 parts; each part has 64 depths and is stored one at a time in the matrix of PEs. It was chosen that the filters are divided into 48 groups of 8 filters.

Those filters will be stored with their corresponding biases in the filter buffer with the whole depth. So, the matrix of the PEs should be 3x13 regarding layer 3.

The stride in this layer is 1. So, after each 1 row of the input is stored, the top PE in the next column is activated for storage to apply the vertical stride. For example, when storing the first row of the input, only one column is activated. When starting to store the second row, the first PE in the second column is activated whilst the first column is still active, meaning that the second row will be written in the second PE of the first column as well as the first PE in the second column. When starting to store the third row, the first PE in the third column is activated whilst the first and second columns are still active, meaning that the third row will be written in the third PE of the first column, as well as the second PE of the second column and the first PE in the third column. This means that the input rows are placed diagonally in the PEs. When the first 3 rows are stored, no other PEs are activated in the same column at this time, as they will be used for the storage of another depth to keep the vertical summation valid, meaning no extra hardware has to be added to that used in the previous. Note that in this layer, 16 depths of the filter and input are stored in the PE each time. After the whole depth of the input is stored, another depth is then stored in the same three rows of PEs (14x3), then the third depth is stored again till 16 depths are stored, after that the second part of PEs (14x3) are activated and the same operation is repeated for this part of PEs. The process is repeated again for the third and fourth parts respectively. The storage of the filter is a straight forward task, as each row of PEs stores the same row of the filter. So, each PE from every part of the 4 parts will contain 16 depths of one filter.

Now, we'll focus on how the convolution operation is performed. As mentioned before, to produce an output pixel, the summations of the multiplications of all the corresponding filter weights and input pixels on all depths have to be computed. This is computed in multiple iterations, as 16 depths only are present in the PE each time. Each PE holds 1 row of the same filter from 16 different depths in one of the RFs and 1 row of the input corresponding to each of those 16 depths in the other RF. The outputs of the convolution of each filter row and input row are summed vertically to produce the partial sums (PSUMs) of one row of one of the output feature maps due to four depths. This operation is repeated 16 times to produce the partial sums (PSUMs) of one row of one of the output feature maps due to 64 depths. Only 13 columns of PEs are needed so the output will be those 13 rows. Here, the depth of each filter is 256 so we still have to store these partial sums in the PSUM buffer. After the operation is completed, another filter with 64 depths is stored in the matrix. This operation is repeated till all the 8 filters are passed on the input. After that, another part of the input containing an additional 64 channels is brought to the PEs, and the first filter with the corresponding 64 depths is brought from the filter buffer. The process is repeated till the 8 filters are passed on this part of the input. After that, the third and fourth parts of the input and filters are stored in the matrix of PEs to compute the final first 16 output feature maps of this layer. Note that during the last time, the input is stored directly to the swapping buffer 1, after adding the bias, and no maxpooling layer is present. Another 8 filters is then brought from the off-chip memory and the whole process is repeated. This process is repeated a total of 48 times, to produce the whole 384 output feature maps.

The size of the PSUM buffer from the point of view of the third layer should be 13x13x16. Where 13 represents the number of rows of each output feature map, 13 represents the size of each row, and 16 represents that the number of filters produced each iteration.

	Storage from the point of view of layer 3
Filter Buffer	3x3x256x16 + 16 = 36880 locations
PSUM Buffer	13x13x16 = 2704 locations
Swapping Buffer 1	13x13x256 = 43264 locations
Swapping Buffer 2	13x13x384 = 64896 locations

4.4.4. Fourth Convolutional layer

The input of this layer is the output of the previous convolutional layer as they are no maxpooling or Local Response Normalization layers, so its size is 13x13x384. It is stored in swapping buffer 1. This layer consists of 384 filters (kernels) each having a size of 3x3x192. The filters are divided into two groups; the first groups consists of the first 192 filters and they get convolved with the first half of the number of depths of the input (i.e. first 13x13x192), the second group consists of the remaining 192 filters and they get convolved with the other half of the number of depths of the input (i.e. last 13x13x192). The output of this layer is 13x13x384. The zero padding technique is used here to ensure that the output feature maps spatial dimensions are the same as those of the input feature maps (i.e. 13x13). As discussed in the row stationary technique, each PE should store one row of the filter as well as one row of the input feature map. Moreover, the accumulation of each row of the output feature map is done vertically. So, to compute one row of the output, 3 PEs should be stacked together vertically as there are 3 rows in each filter.

Due to the presence of the two groups, the input is divided into 2 parts; each part consists of 192 depths. The first part will produce the first 192 output feature maps when convolved with the first 192 filters, and so the second part will produce the remaining 192 feature maps. It was chosen that the filters of each part are divided into 12 batches of 16 filters each. Those filters will be stored with their corresponding biases in the filter buffer. So, the matrix of the PEs should be 3x13 regarding layer 4.

The stride in this layer is 1. So, after each 1 row of the input is stored, the top PE in the next column is activated for storage to apply the vertical stride. For example, when storing the first row of the input, only one column is activated. When starting to store the second row, the first PE in the second column is activated whilst the first column is still active, meaning that the second row will be written in the second PE of the first column as well as the first PE in the second column. When starting to store the third row, the first PE in the third column is activated whilst the first and second columns are still active, meaning that the third row will be written in the third PE of the first column, as well as the second PE of the second column and the first PE in the third column. This means that the input rows are placed diagonally in the PEs. When the first 3 rows are stored, no other PEs are activated in the same column at this time, as they will be used for the storage of another depth to keep the vertical summation valid, meaning no extra hardware has to be added to that used in layer 1. Note that in this layer, 16 depths of the filter and input are stored in the PE each time. After the whole depth of the input is stored, another depth is then stored in the same three rows of PEs (14x3), then the third depth is stored again till 16 depths are stored, after that the second part of PEs (14x3) are activated and the same operation is repeated for this part of PEs. The process is repeated again for the third and fourth parts respectively. The storage of the filter is a straight forward task, as each row of PEs stores the same row of the filter. So, each PE from every part of the 4 parts will contain 16 depths of one filter.

Now, we'll focus on how the convolution operation is performed. As mentioned before, to produce an output pixel, the summations of the multiplications of all the corresponding filter weights and input pixels on all depths have to be computed. This is computed in multiple iterations, as 16 depths only are present in the PE each time. Each PE holds 1 row of the same filter from 16 different depths in one of the RFs and 1 row of the input corresponding to each of those 16 depths in the other RF. The outputs of the convolution of each filter row and input row are summed vertically to produce the partial sums (PSUMs) of one row of one of the output feature maps due to four depths. This operation is repeated 16 times to produce the partial sums (PSUMs) of one row of one of the output feature maps due to 64 depths. Only 13 columns of PEs are needed so the output will be those 13 rows. Here, the depth of each filter of a group is 192 so we still have to store these partial sums in the PSUM buffer. After the operation is completed, another filter with 64 depths is stored in the matrix. This operation is repeated till all the 16 filters are passed on the input. After that, another part of the input containing an additional 64 channels is brought to the PEs, and the first filter with the corresponding 64 depths is brought from the filter buffer. The process is repeated till the 16 filters are passed on this part of the input. After that, the third part of the input and filters is stored in the matrix of PEs to compute the final first 16 output feature maps of this layer. Note that during the last time, the input is stored directly to the swapping buffer 2, after adding the bias, and no maxpooling layer is present. Another 16 filters is then brought from the off-chip memory and the whole process is repeated. This process is repeated a total of 12 times, to produce the first 192 output feature maps. Then, this part of the input isn't needed anymore, so the same operation is performed with the other part of the input till the other 192 output feature maps are computed.

The size of the PSUM buffer from the point of view of the fourth layer should be 13x13x16. Where 13 represents the number of rows of each output feature map, 13 represents the size of each row, and 16 represents that the number of filters produced each iteration.

	Storage from the point of view of layer 4
Filter Buffer	3x3x192x16 + 16 = 27664 locations
PSUM Buffer	13x13x16 = 2704 locations
Swapping Buffer 1	13x13x384 = 64896 locations
Swapping Buffer 2	13x13x384 = 64896 locations

4.4.5. Fifth Convolutional Layer

The input of this layer is the output of the previous convolutional layer as they are no maxpooling or Local Response Normalization layers, so its size is 13x13x384. It is stored in swapping buffer 2. This layer consists of 256 filters (kernels) each having a size of 3x3x192. The filters are divided into two groups; the first groups consists of the first 128 filters and they get convolved with the first half of the number of depths of the input (i.e. first 13x13x192), the second group consists of the remaining 128 filters and they get convolved with the other half of the number of depths of the input (i.e. last 13x13x192). The output of this layer is 13x13x384. The zero padding technique is used here to ensure that the output feature maps spatial dimensions are the same as those of the input feature maps (i.e. 13x13). As discussed in the row stationary technique, each PE should store one row of the filter as well as one row of the input feature map. Moreover, the accumulation of each row of the output feature map is done vertically. So, to compute one row of the output, 3 PEs should be stacked together vertically as there are 3 rows in each filter.

Due to the presence of the two groups, the input is divided into 2 parts; each part consists of 192 depths. The first part will produce the first 128 output feature maps when convolved with the first 128 filters, and so the second part will produce the remaining 128 feature maps. It was chosen that the filters of each part are divided into 8 batches of 16 filters each. Those filters will be stored with their corresponding biases in the filter buffer. So, the matrix of the PEs should be 3x13 regarding layer 5.

The stride in this layer is 1. So, after each 1 row of the input is stored, the top PE in the next column is activated for storage to apply the vertical stride. For example, when storing the first row of the input, only one column is activated. When starting to store the second row, the first PE in the second column is activated whilst the first column is still active, meaning that the second row will be written in the second PE of the first column as well as the first PE in the second column. When starting to store the third row, the first PE in the third column is activated whilst the first and second columns are still active, meaning that the third row will be written in the third PE of the first column, as well as the second PE of the second column and the first PE in the third column. This means that the input rows are placed diagonally in the PEs. When the first 3 rows are stored, no other PEs are activated in the same column at this time, as they will be used for the storage of another depth to keep the vertical summation valid, meaning no extra hardware has to be added to that used in layer 1. Note that in this layer, 16 depths of the filter and input are stored in the PE each time. After the whole depth of the input is stored, another depth is then stored in the same three rows of PEs (14x3), then the third depth is stored again till 16 depths are stored, after that the second part of PEs (14x3) are activated and the same operation is repeated for this part of PEs. The process is repeated again for the third and fourth parts respectively. The storage of the filter is a straight forward task, as each row of PEs stores the same row of the filter. So, each PE from every part of the 4 parts will contain 16 depths of one filter.

Now, we'll focus on how the convolution operation is performed. As mentioned before, to produce an output pixel, the summations of the multiplications of all the corresponding filter weights and input pixels on all depths have to be computed. This is computed in multiple iterations, as 16 depths only are present in the PE each time. Each PE holds 1 row of the same filter from 16 different depths in one of the RFs and 1 row of the input corresponding to each of those 16 depths in the other RF. The outputs of the convolution of each filter row and input row are summed vertically to produce the partial sums (PSUMs) of one row of one of the output feature maps due to four depths. This operation is repeated 16 times to produce the partial sums (PSUMs) of one row of one of the output feature maps due to 64 depths. Only 13 columns of PEs are needed so the output will be those 13 rows. Here, the depth of each filter of a group is 128 so we still have to store these partial sums in the PSUM buffer. After the operation is completed, another filter with 64 depths is stored in the matrix. This operation is repeated till all the 16 filters are passed on the input. After that, another part of the input containing an additional 64 channels is brought to the PEs, and the first filter with the corresponding 64 depths is brought from the filter buffer. The process is repeated till the 16 filters are passed on this part of the input. The bias is then added to the output

PSUM. Note that between the finish of the 16 filters in a batch and the storage of another 16, the maxpooling layer operates to store the outputs in the swapping buffer 1 again. The maxpooling layer will be discussed in the next section. Another 16 filters is then brought from the off-chip memory and the whole process is repeated. This process is repeated a total of 8 times, to produce the first 128 output feature maps. Then, this part of the input isn't needed anymore, so the same operation is performed with the other part of the input till the other 128 output feature maps are computed.

The size of the PSUM buffer from the point of view of the fifth layer should be 13x13x16. Where 13 represents the number of rows of each output feature map, 13 represents the size of each row, and 16 represents that the number of filters produced each iteration.

	Storage from the point of view of layer 5
Filter Buffer	3x3x192x16 + 16 = 27664 locations
PSUM Buffer	13x13x16 = 2704 locations
Swapping Buffer 1	----
Swapping Buffer 2	13x13x384 = 64896 locations

4.4.6. Utilization of Resources

4.4.6.1. PE Matrix

Table 1 shows the PE matrix from the point of view of each layer

layer	row	column	PE matrix
1	11	14	11x14
2	10	14	10x14
3	12	13	12x13
4	12	13	12x13
5	12	13	12x13

The best choice of the PE matrix would be 14x12 PEs

Layer	Utilization of PEs (%)
1	154(91.66%)
2	140(83.33%)
3	156(92.85%)
4	156(92.85%)
5	156(92.85%)

* These calculations is based on the normal operation of each layer and not taking into consideration any special cases.

4.4.6.2. MAC Operations

Table 2 shows the number of MAC operations for each layer.

Layer	MAC operations (MOPS)
1	105.41 M
2	447.79 M
3	415.33 M
4	623.00 M
5	415.33 M

4.4.6.3. Latency

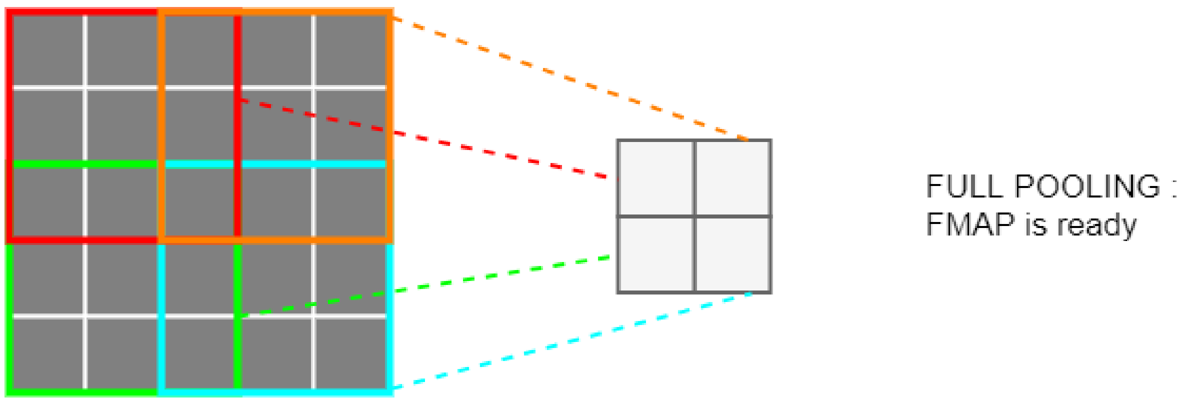
Table 3 shows the latency for each layer including the buffer access and the convolution operations.

Layer	Latency(milliseconds)
1	41.1
2	60.8
3	78.8
4	58.0
5	43.5

4.5. Pooling Layers

There are three maxpooling layers (pool1, pool2 and pool5) present in our design. Each pooling layer has a window of 3x3 and a stride of 2. The main requirement for the operation of this pooling layer is that the pooling layer should have the whole feature map of any depth ready to perform the pooling successfully. This is valid in the second through fifth layers as the sizes of the feature maps are relatively small in size compared to the first layer. Max pooling for these layers are shown in figure 2.

-



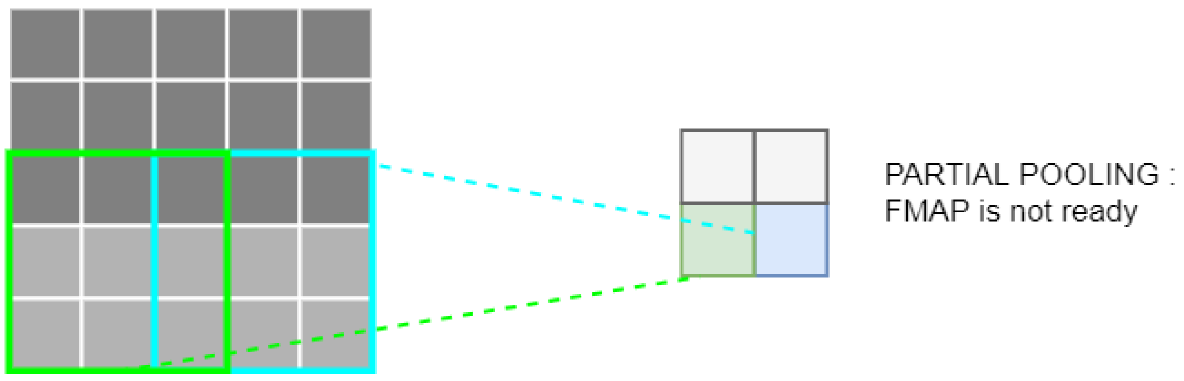
So, this requirement is difficult to fulfill in the first layer for three main reasons:

1. The feature map is large in size (55x55).

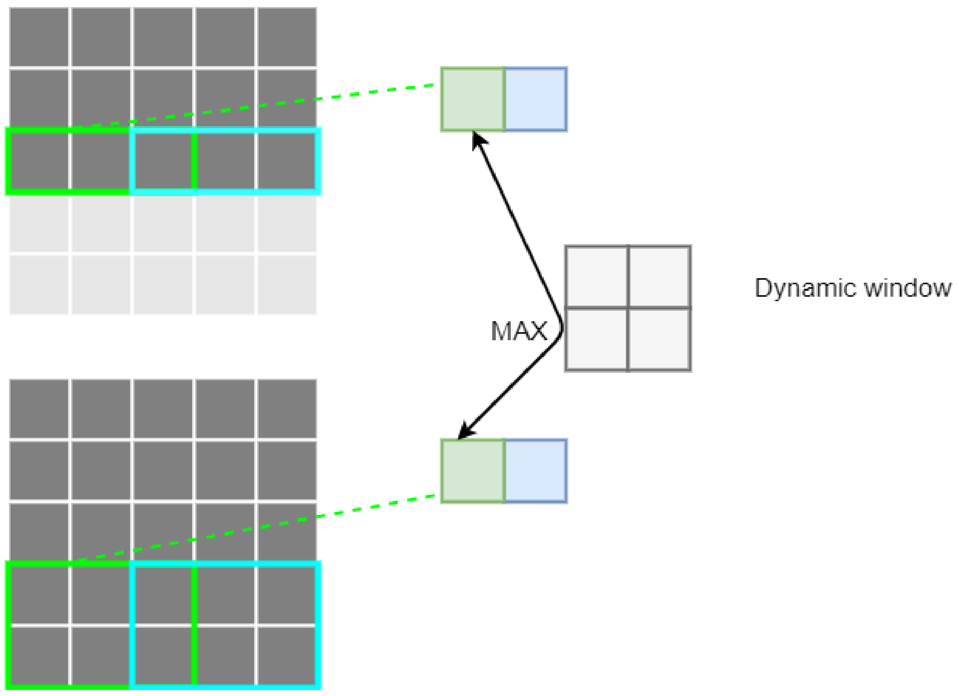
2. The size of the global buffers would be massive in order to store all these feature maps.

3. This will cause much unwanted delays.

The solution is that the convolution is divided into several iterations to accommodate for these problems. The maxpooling layer receives nearly a quarter of a feature map only from the convolution block each iteration. This created another problem of not being able to stride in the vertical direction in the last 2 rows of the part of the feature map present as shown in figure2.



To solve this dilemma, partial pooling technique, which uses a dynamic window, was performed for the rows facing the same problem. The pooling is divided into two parts; the first part contains the maxpooling of the last two rows. This is equivalent to a 2x3 window, so we still need the remaining 1x3 to complete the whole 3x3 window. The remaining 1x3 is taken from the first row from the part of the feature map ready during the next iteration. The last step is to choose from the maximum of both parts. This is illustrated in figure3 below.



Now, each pooling layer will be discussed in details.

4.5.1. Pooling Layer 1

The input of this layer is the output of the first convolutional layer. This layer takes the output of the convolution after each convolution iteration, and performs the pooling operation on it. So, the total input of this layer is 55x55x96, but each time only 14x55x16 is available. The total output of this layer is 27x27x96 but each iteration it porduces 7x27x16. The window is 3x3 and the stride is 2 as mentioned above. The problem of incomplete feature map is present here, so the partial pooling technique is used here. This layer operates for 24 times.

The operation of this layer is based on the sliding of the pooling window, while keeping into considerations that there is a jump every 3 pixels of the input to get the residing pixels in the next two rows. This maxpool layer takes the data from the PSUM buffer to the switching 1 buffer.

4.5.2. Pooling Layer 2

The input of this layer is the output of the second convolutional layer. This layer takes the output of the convolution after each convolution iteration, and performs the pooling operation on it. So, the total input of this layer is 27x27x256, but each time only 27x27x15 is available, note that sometimes convolutional layer 2 outputs only 27x27x8. The total output of this layer is 13x13x256 but each iteration it produces 13x13x15. The window is 3x3 and the stride is 2 as mentioned above. The problem of incomplete feature map is not present here, so full pooling technique is used here.

The operation of this layer is based on the sliding of the pooling window, while keeping into considerations that there is a jump every 3 pixels of the input to get the residing pixels in the next two rows. This maxpool layer takes the data from the PSUM buffer to the switching 1 buffer as well as layer 1.

4.5.3. Pooling Layer 5

The input of this layer is the output of the fifth convolutional layer. This layer takes the output of the convolution after each convolution iteration, and performs the pooling operation on it. So, the total input of this layer is 13x13x256, but each time only 13x13x16 is available. The total output of this layer is 6x6x256 but each iteration it porduces 6x6x16. The window is 3x3 and the stride is 2 as mentioned above.

The operation of this layer is based on the sliding of the pooling window, while keeping into considerations that there is a jump every 3 pixels of the input to get the residing pixels in the next two rows. This maxpool layer takes the data from the PSUM buffer to the switching 1 buffer.

4.5.4. Utilization of resources

4.5.4.1. Latency

Table 3 shows the latency for each layer including the buffer access and the convolution operations.

Layer	Latency(milliseconds)
1	14
2	8.65
5	1.84

4.6. Local Response Normalization (LRN) Layers

In this section, the hardware implementation of the normalization formula will be discussed. As mentioned before in section (), the computation of the normalization layer can be formulated as follows:

$$b_{x,y}^{i} = \frac{a_{x,y}^{i}}{\left(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a_{x,y}^{j})^2\right)^{\beta}}$$

Where the terms $a_{x,y}^{i}$ and $b_{x,y}^{i}$ are input and output neurons at location <x , y> of feature map i, respectively. N refers to the total number of feature maps in the layer and n refers to the number of adjacent feature maps to be considered.

Each excited neuron (pixel) is normalized around its neighbourhood through the depth of the feature maps, so it becomes even more sensitive as compared to its neighbours. Fig.() illustrates the normalization operation through the depth of the feature maps.

It is clear that the formula is composed mainly of adders, multipliers and division blocks. The schematic is shown in Fig.() describing the mathematical flow of the normalization formula. Each block in the schematic is terminated by a register to store the output of the block for duration of one clock cycle. The advantage of this architecture is to break the long 1 cycle path of the block to a multi-cycle path with a much less cycle period, thus, increasing the frequency of the used clock. Another advantage is to allow the use of pipelining through this multi-cycle path, thus, boosting the throughput of the whole block by 9 times.

The multi-cycle path of the block consists of 18 cycles; 2 cycles for the input register and the first adder, respectively. The remaining 16 cycles are for the division block. The adders and multipliers used are implemented automatically as LUTs and DSPs, respectively, on the FPGA. While the division block is more complicated and will be discussed in the next sub-section.

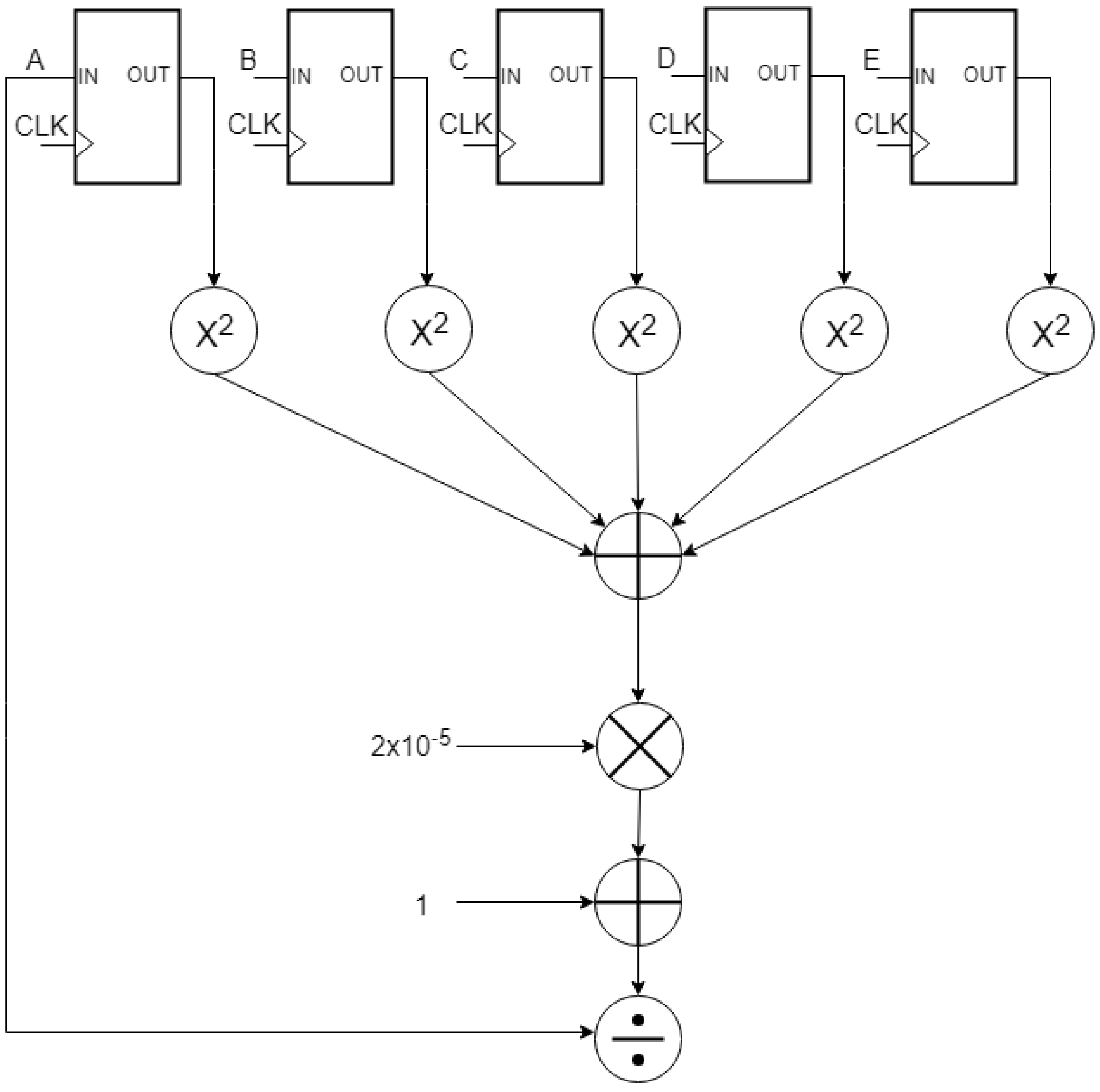
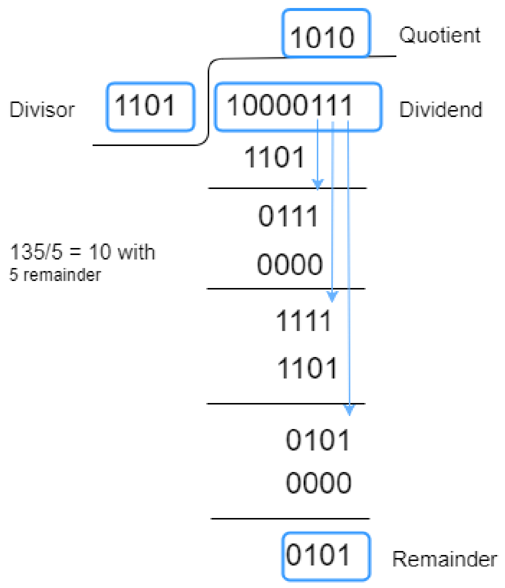


Figure 22. LRN schematic

4.6.1. Division Block

When you write / you're asking for a divider circuit, but a divider circuit is a very complex thing. It often takes multiple clock cycles, and may use look up tables. It's asking a lot of a synthesis tool to infer what you want when you write a / b. The same is true of *, but to a lesser degree. Multipliers are quite expensive, but most synthesisers are able to infer them. So that’s why we had to design a synthesizable division circuit as illustrated in the next section.

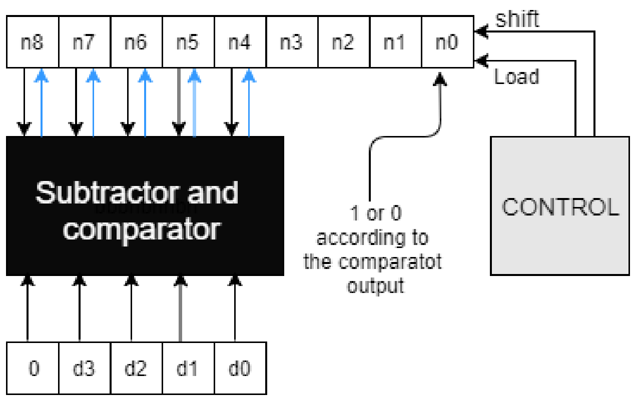
The following example shows how to do divide two binary numbers, and this is the base of our design.



As depicted from the above example, the division algorithm repeatedly subtracts the divisor (multiplied by one or zero) from appropriate bits of the dividend. Therefore, subtraction and shift operations are the two basic operations to implement the division algorithm.

After each subtraction, the divisor (multiplied by one or zero) is shifted to the right by one bit relative to the dividend. For the circuit implementation, we will shift the dividend to the left rather than shifting the divisor to the right as shifting left requires more registers. Besides, the numerical example shows that, as we proceed with the algorithm, some significant bits of the dividend term are no longer required and can be discarded.

A simplified block diagram for dividing an eight-bit number by a four-bit number is shown in Figure 2. The nine-bit register, n_8 , n7 ,…..,n0 stores the value of the dividend and the four-bit register, d3,d2,…,d0, is used to store the divisor.
n0 is the extra bit which will be used to store the bit of the Quotient each iteration. At the beginning of the algorithm, this bit is set to zero. The flow chart of the division algorithm is illustrated in figure3.


n8
n7
n6
n5
n4
n3
n2
n1
n0
shift
Load
Subtractor and comparator
1 or 0 according to the comparatot output
CONTROL
0
d3
d2
d1
d0

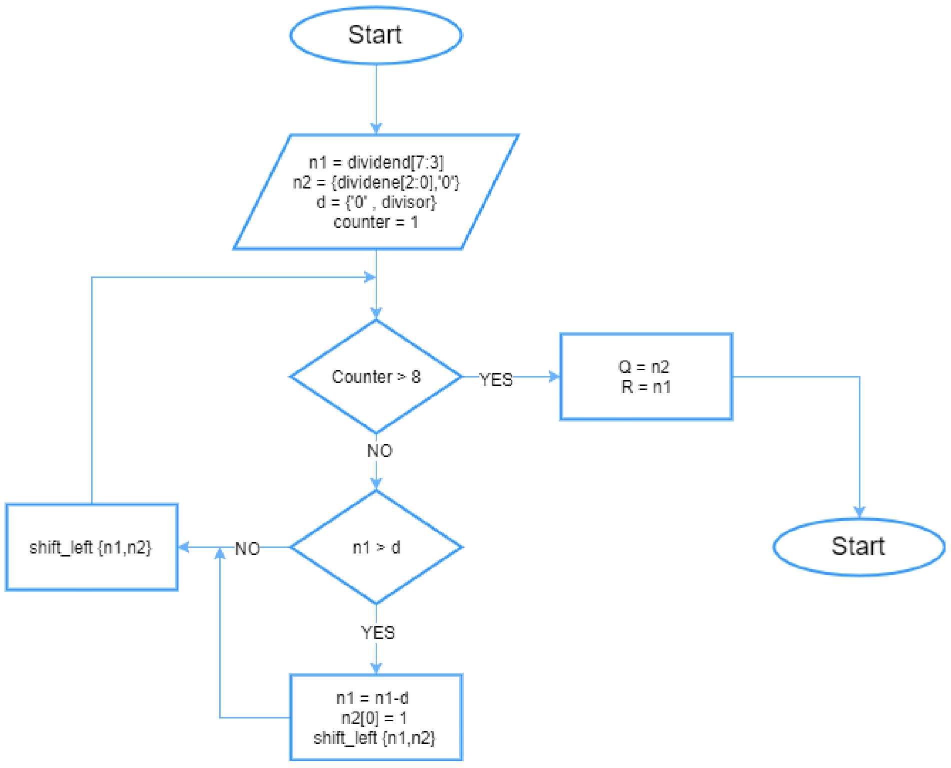
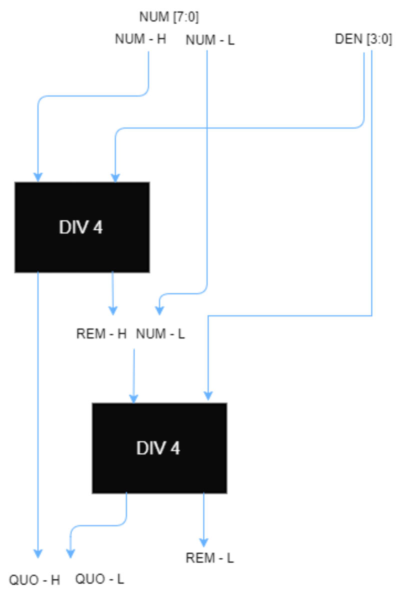


Figure 23. Division Algorithm

Overflow problem:

At the beginning of the algorithm, we must have n8n7n6n5<d3d2d1d0, otherwise, the quotient will be greater than 1111_2=15_10 and we cannot represent it in the vacated locations of the N(numerator) register. To solve that we can do the division on two stages as shown in fig .



This design is for 8-bit division, it can be scaled up to 16, 32, etc...

4.7. Fully Connected Layers

4.7.1. Reshaping

The input to the fully connected layers needs to be a vector so reshape is done to the output of the POOL5 layer (6x6x256) to make it a vector of 9216x1.

4.7.2. Fully Connected 6

The input size of this layer is 9216 pixels, which is stored in one of the swapping buffers. There are 4096 filters (neurons) each of 9216 weights, 2 filters are stored in the other swapping buffer as well as the 4096 biases.

When storing in the PEs, the caches are divided into two groups. The first group store the input pixels where each cache is filled with 256 pixels of the input pixels so the whole input needs 36 PEs (distributed among 6 columns) to be stored, hence the input is stored 2 times to exploit a total of 72 PEs (12 columns) except for the last 2 columns of PEs which are not used. The second group store the filter pixels where each cache is filled with 256 weights of the filter so the whole filter needs 36 PEs (distributed among 6 columns) to be stored, hence two filters are stored each time in the even buffer.

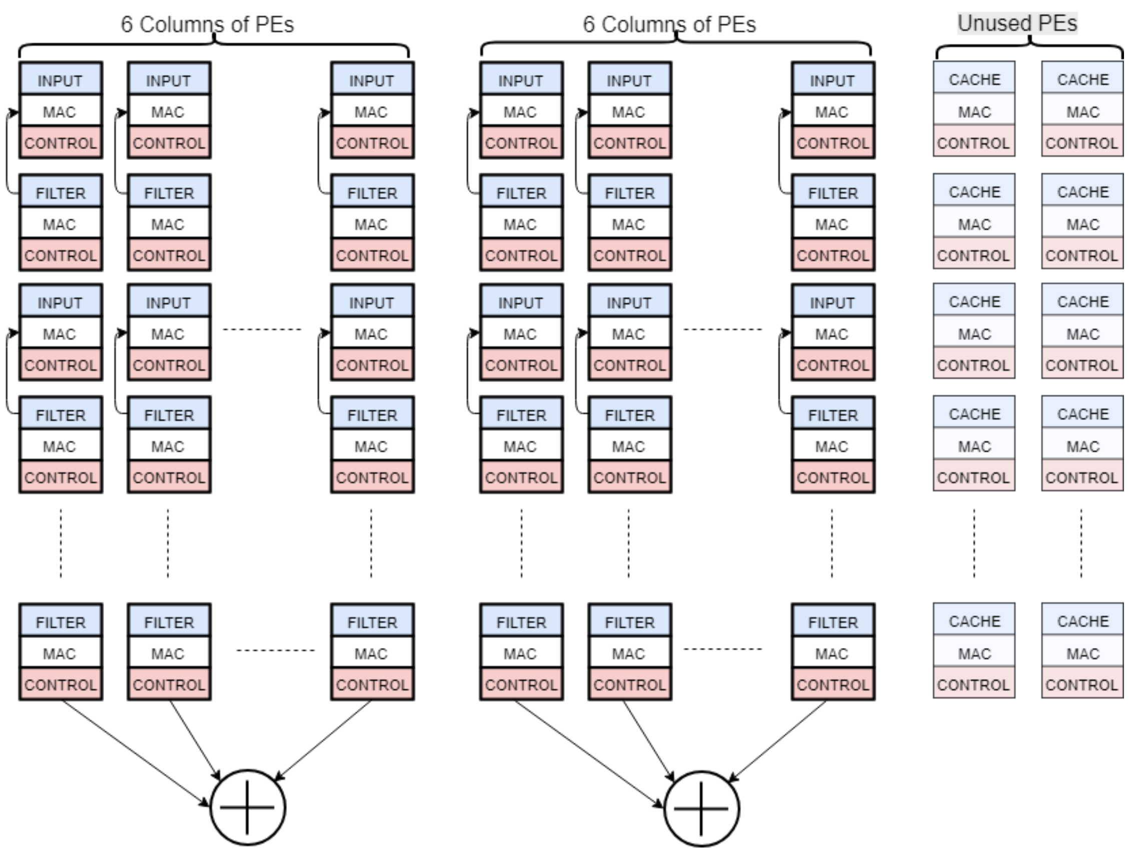


Figure 24. Fully Connected 6

After MAC operation between input and filters, the bias is added to the output and stored in one of the swapping buffers to be the input of the next layer, then the control unit requests a new filter batch of size equals 2 to be transferred from the external memory to the other swapping buffer and the previous operation is repeated after receiving a start signal to initiate it again, so as to assure that the filter batch has been written in the buffer.

4.7.3. Fully Connected 7

The input size of this layer is 4096 pixels, the input is stored in the swapping buffer mentioned before. There are 4096 filters each of 4096 pixels, only 3 filters are stored in the swapping buffer as well as the 4096 biases.

Each feature map cache is filled with 171 pixels of input pixels, so the whole input needs 24 PEs (distributed among 4 columns) to be stored but the last feature map cache of the 24 PEs is only filled with 163 pixels of input, hence the input is stored 3 times to exploit a total of 144 PEs (12 columns).

Each filter cache is also filled with 171 pixels of a filter so each filter is stored in 4 columns as well but the last filter cache of the 24 PEs is only filled with 163 pixels of that filter, hence a total of 3 filters are stored to be multiplied with the input.

After MAC operation between input and filters, the bias is added to the output and stored in one of the swapping buffers to be the input of the next layer, then the control unit requests a new filter batch of size equals 2 to be transferred from the external memory to the other swapping buffer and the previous operation is repeated after receiving a start signal to initiate it again, so as to assure that the filter batch has been written in the buffer.

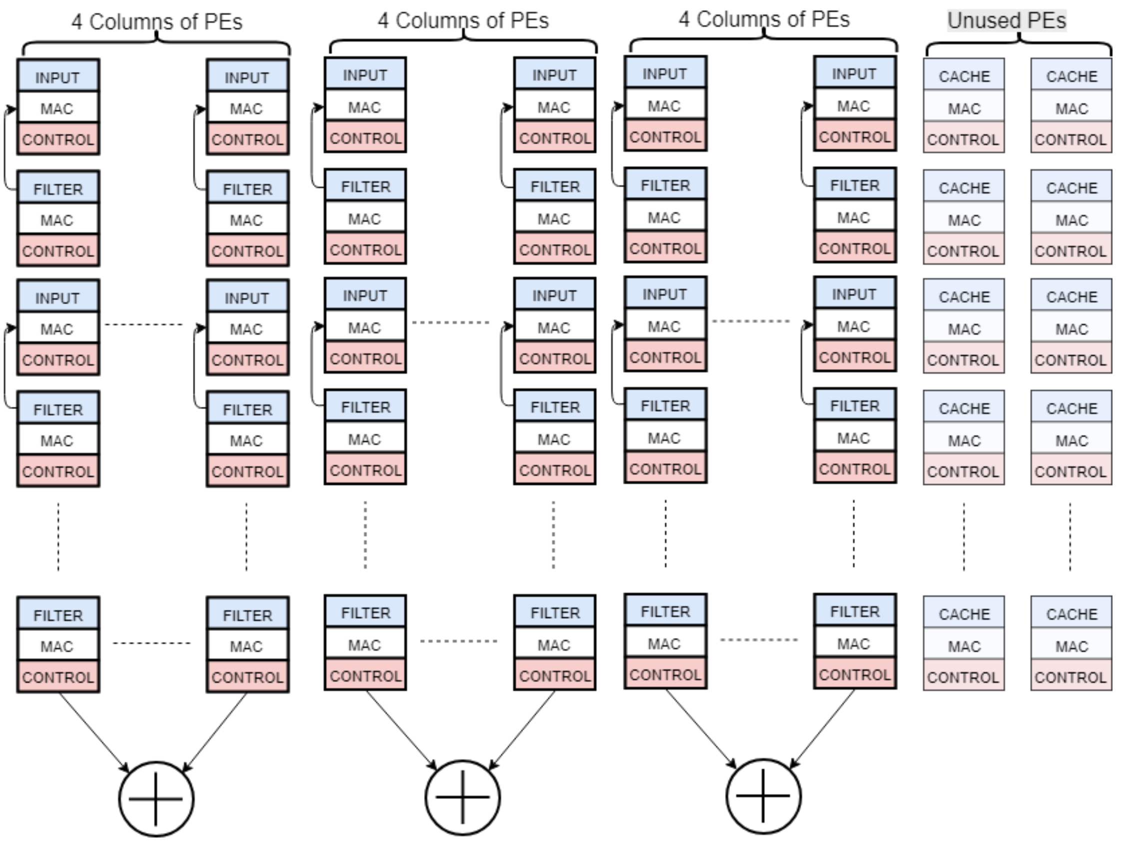


Figure 25. Fully Connected 7 & 8

4.7.4. Fully Connected 8

Fully Connected 8 is exactly the same in its operation as Fully Connected 7 except that the number of filters here is 1000 filters (neurons) instead of 4096 in Fully Connected 7, so the same procedure is done.

It computes the class probability scores by outputting a vector of 1000 dimensions, where 1000 being the number of classes.

Estimation is done to the output of this layer to determine the class of the input which is the class with the highest probability score.

4.8. Memory Hierarchy

Deep learning using convolutional neural networks (CNNs) has achieved unprecedented accuracy on many modern AI applications. However, state-of-the art CNNs require tens to hundreds of megabytes of parameters on billions of operations in a single inference pass, creating significant data movement from on-chip and off-chip to support the computation. Since data movement can be more energy-consuming than computation, the processing of CNNs has to not only provide high parallelism for high throughput but also optimize for the data movement of the entire system in order to achieve high energy efficiency.
In addition, this optimization needs to adapt to the varying shapes of the high-dimensional convolutions in CNN. To address these challenges, it is crucial to design a compute scheme, called a *dataflow*, which can support a highly parallel compute paradigm while optimizing the energy cost of data movement from both on-chip and off-chip. The cost of data movement is reduced by exploiting data reuse in a multilevel memory hierarchy, and the hardware needs to be reconfigurable to support different shapes.

CNN shapes, including square and non-square filters. The presence of the PEs creates a three-level memory hierarchy as shown in fig.().
Data movement can exploit the low-cost levels, such as the PE scratch pads (spads) and the inter-PE communication, to minimize data accesses to the high-cost levels, including the large on-chip global buffer (GLB) and the off-chip DRAM.

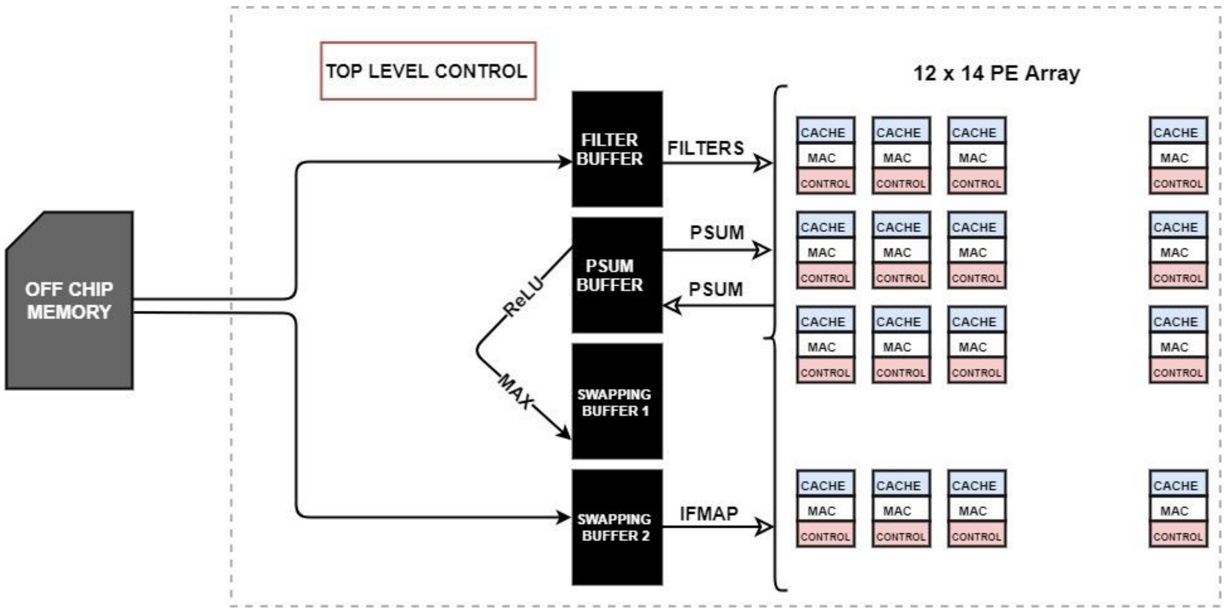
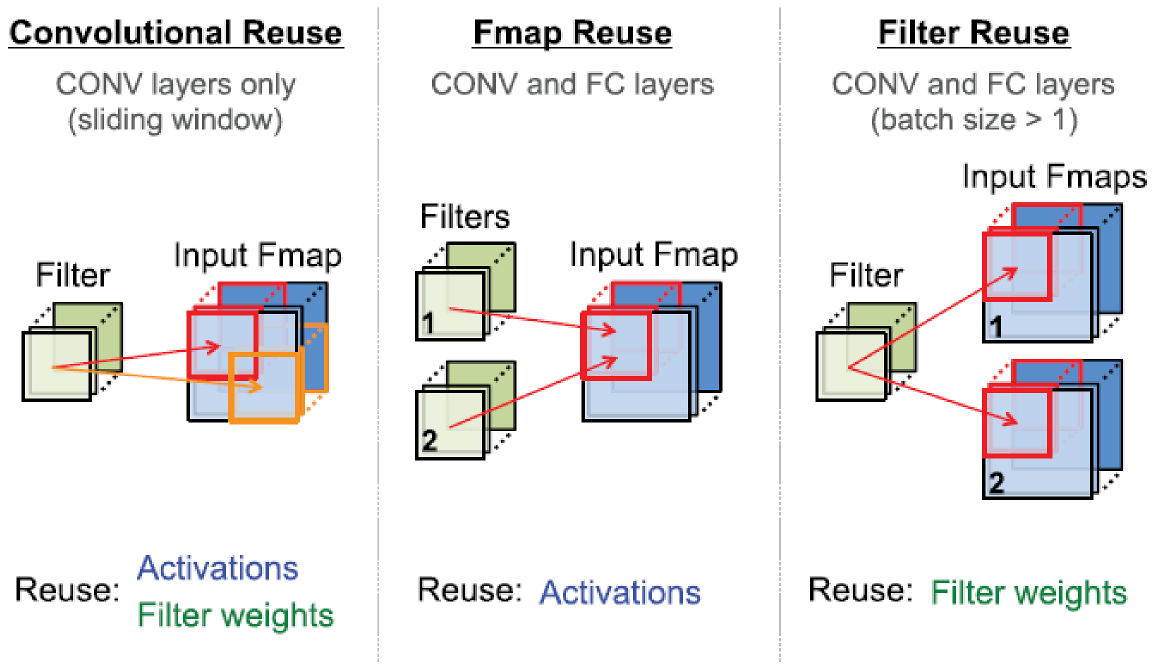


Figure 26. Memory Hierarchy

Each MAC requires three memory reads (for filter weight, FMAP activation, and partial sum) and one memory write (for the updated partial sum). In the worst case, all of the memory accesses have to go through the off-chip DRAM, which will severely impact both throughput and energy efficiency. For example, in AlexNet, to support its 724 million MACs, nearly 3000 million DRAM accesses will be required. Furthermore, DRAM accesses require up to several orders of magnitude higher energy than computation, as large memories that can store a significant amount of data consume more energy than smaller memories. For instance, DRAM can store gigabytes of data, but consumes two orders of magnitude higher energy per access than a small on-chip memory of a few kilobytes. Thus, every time a piece of data is moved from an expensive level to a lower cost level in terms of energy, we want to reuse that piece of data as much as possible to minimize subsequent accesses to the expensive levels. The challenge, however, is that the storage capacity of these low cost memories is limited. Thus we need to explore different data flows that maximize reuse under these constraints. For DNNs, we investigate data flows that exploit three forms of input data reuse (convolutional, FMAP, and filter) as shown in figure(). 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. If we can harness the three types of data reuse by storing the data in the local memory hierarchy and accessing them multiple times without going back to the DRAM, it can save a significant amount of DRAM accesses. For example, in AlexNet, the number of DRAM reads can be reduced by up to 500 × in the CONV layers. The local memory can also be used for partial sum accumulation, so they do not have to reach DRAM. In the best case, if all data reuse and accumulation can be achieved by the local memory hierarchy, the 3000 million DRAM accesses in AlexNet can be reduced to only 61 million.

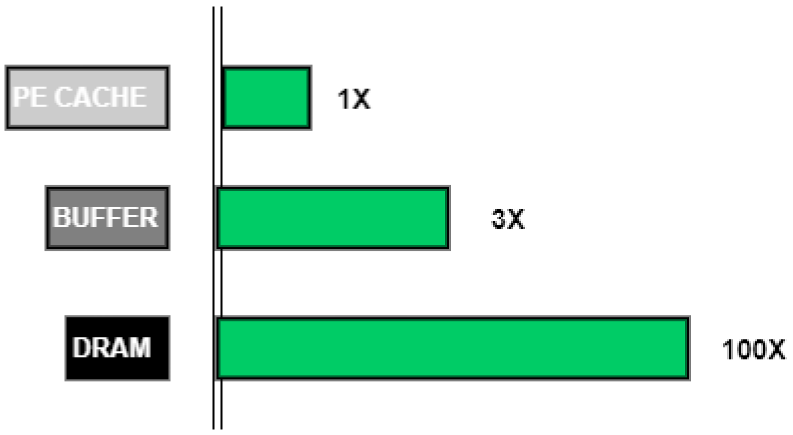


Overall, there are three levels of memory hierarchy in the system (in order of decreasing energy per access):

- DRAM (SD card)
- GLOBAL BUFFERS (FILTER, PSUM, SWAPPING 1, and SWAPPING 2)
- Inter-PE CACHES.

The following table has descriptions for each of them.

Memory		**Function**	**Size**
SD Card		Stores all the weights and biases of the network	256 MB
GLOBAL BUFFERS	FILTER	Stores the current weights needed for convolution	64 KB
	PSUM	Stores the output of the current convolution iteration	32 KB
	SWAPPING 1	Stores the output of some of the convolution layers	256 KB
	SWAPPING 2	Stores the output of some of the convolution layers	256 KB
PE Caches		Stores the part of the input and weights for local reuse	0.5 KB



4.9. Co-Design of DNN Models and Hardware

In earlier work, the DNN models were designed to maximize accuracy without much consideration of the implementation complexity. However, this can lead to designs that are challenging to implement and deploy. To address this, recent work has shown that DNN models and hardware can be co-designed to jointly maximize accuracy and throughput, while minimizing energy and cost, which increases the likelihood of adoption. In this section, we will highlight various efforts that have been made toward the co-design of DNN models and hardware. The techniques discussed in this section can affect the accuracy; thus, the goal is to not only substantially reduce energy consumption and increase throughput, but also to minimize any degradation in accuracy. The co-design approach can be loosely grouped into the following category:

• Reduce precision of operations and operands (this includes going from floating point to fixed point, reducing the bitwidth, non-uniform quantization, and weight sharing).

4.9.1. Reduce Precision

Quantization involves mapping data to a smaller set of quantization levels. The ultimate goal is to minimize the error between the reconstructed data from the quantization levels and the original data. The number of quantization levels reflects the precision and ultimately the number of bits required to represent the data (usually log 2 of the number of levels); thus, reduced precision refers to reducing the number of levels, and thus the number of bits. The benefits of reduced precision include reduced storage cost and/or reduced computation requirements. There are several ways to map the data to quantization levels. The simplest method is a mapping with uniform distance between each quantization level. Another approach is to use a simple mapping function such as a log function where the distance between the levels varies; this mapping can often be implemented with simple logic such as a shift. Alternatively, a more complex mapping function can be used where the quantization levels are determined or learned from the data, e.g., using *k* -means clustering; for this approach, the mapping is usually implemented with a lookup table. Finally, the quantization can be fixed (i.e., the same method of quantization is used for all data types and layers, filters, and channels in the network); or it can be variable (i.e., different methods of quantization can be used for weights and activations, and different layers, filters, and channels in the network). Reduced precision research initially focused on reducing the precision of the weights rather than the activations, since weights directly increase the storage capacity requirement, while the impact of activations on storage capacity depends on the network architecture and dataflow. However, more recent works have also started to look at the impact of quantization on activations. Most reduced precision research also focus on reducing the precision for inference rather than training due to the sensitivity of the gradients to quantization. Fig.() tabulates the accuracy of the software pretrained AlexNet accuracy when different precisions were used.

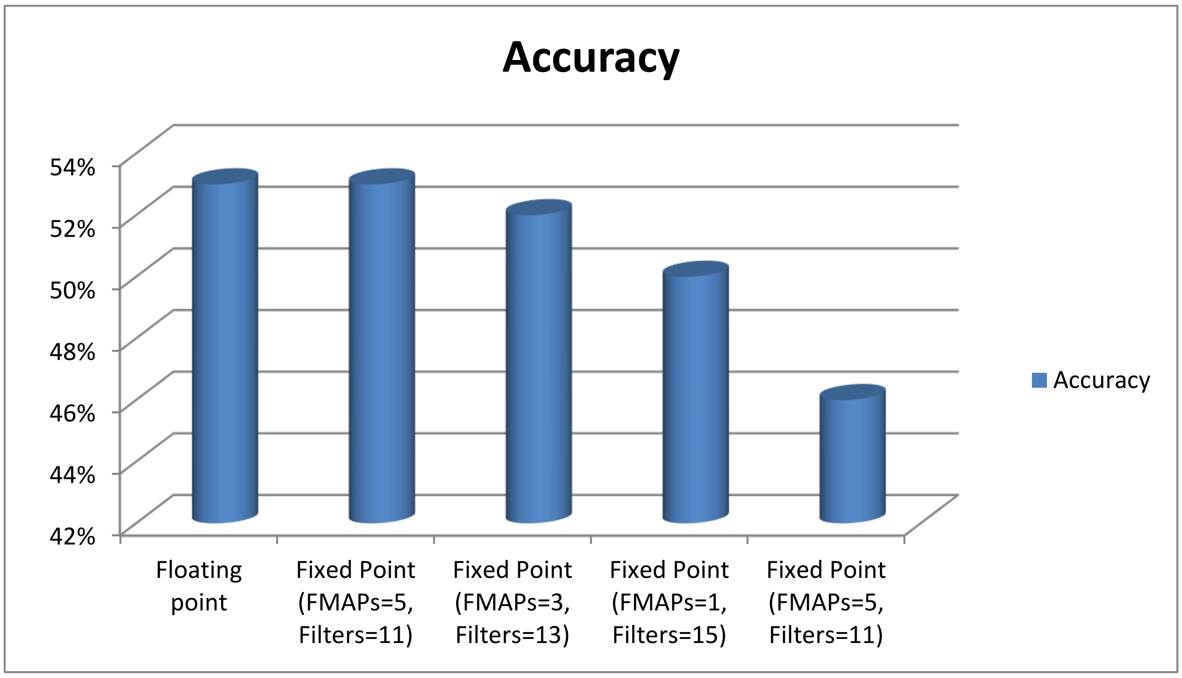


Figure 27. Accuracy vs Precision

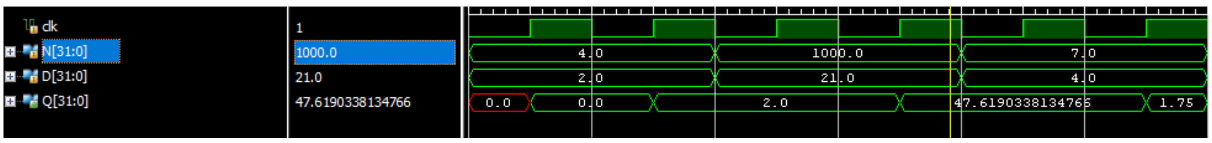
Chapter 5 : Results and Discussion

5.1. Synthesized and Operable Blocks on FPGA

Zynq ZC-702 FPGA was used to implement some of the major blocks of the network. The implemented blocks include the estimation block, the second maxpool layer and the fifth maxpool layer, and their results were compared to those of MATLAB and Vivado.

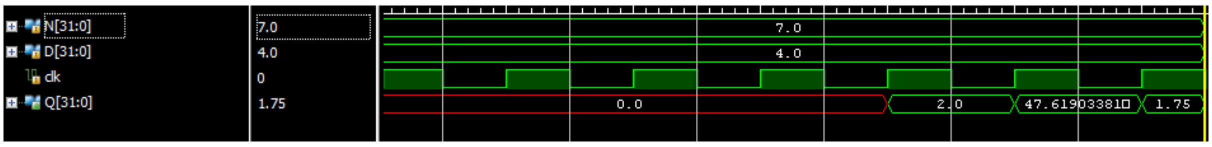
5.2. Division Block

In our design we used 32-bits fixed point division block. Figure shows the simulations, and the following table shows the power and time calculations.



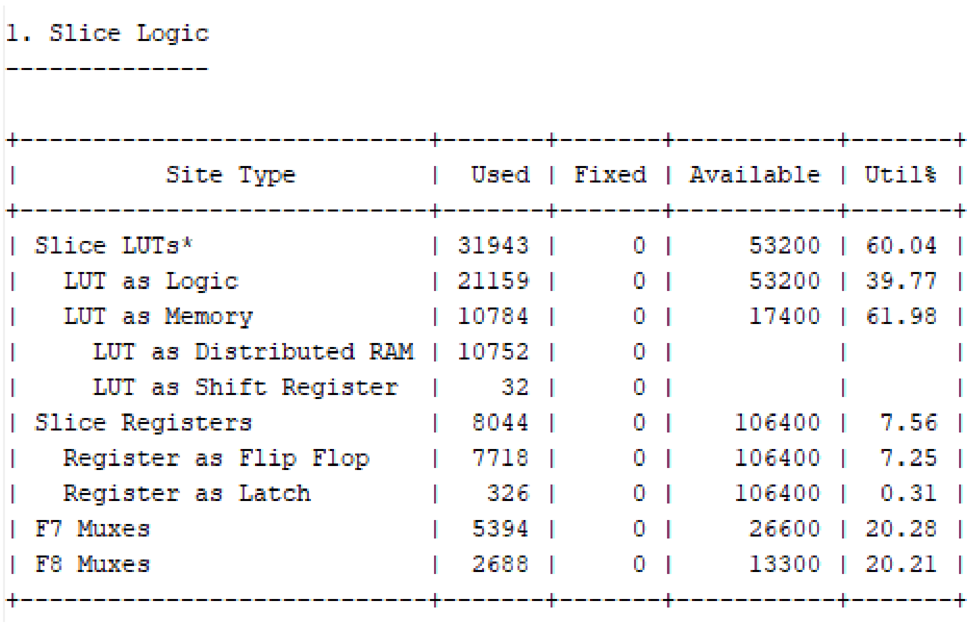
Maximum Frequency	4.88 MHz
Dynamic Power(50% switching)	7 mW
Static Power	120 mW

This design seems to be very slow for AI needs, so pipelining technique is used to get it faster. Figure shows pipelining simulations, and table shows the power and time calculations



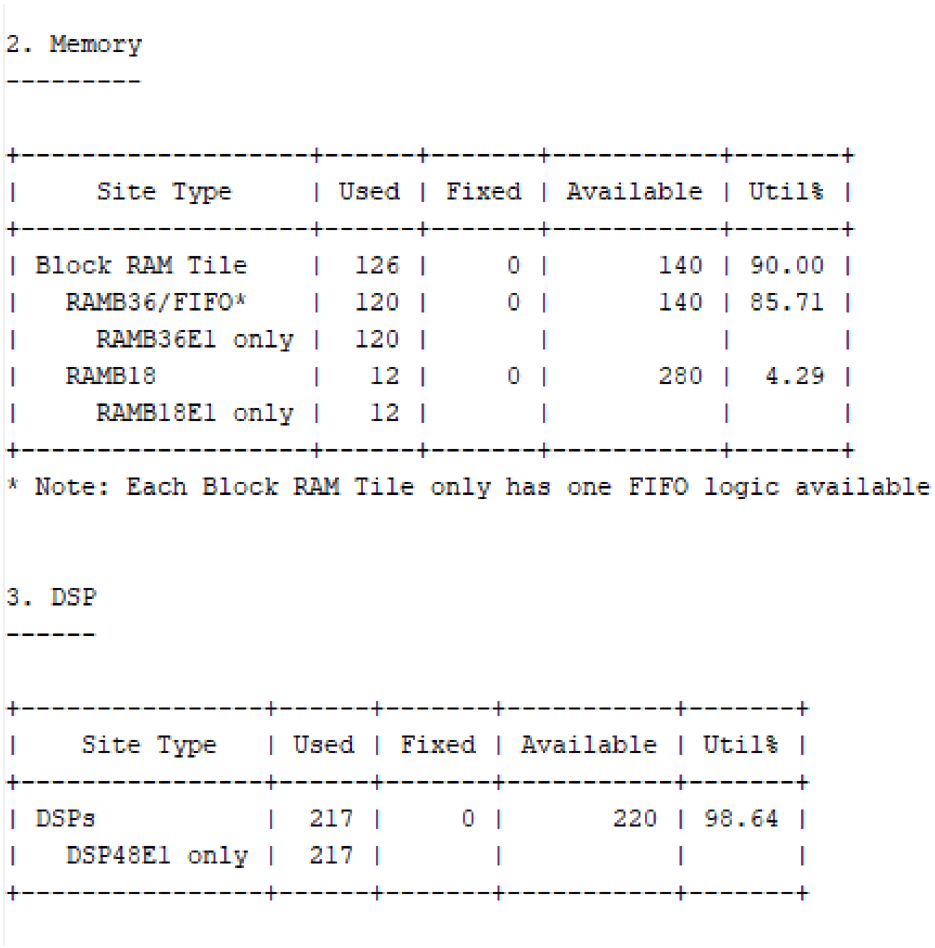
Maximum Frequency	59.33 MHz
Dynamic Power(50% switching)	79 mW
Static Power	121 mW

5.3. FPGA Utilization



```
1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs*                | 31943 |     0 |     53200 | 60.04 |
|   LUT as Logic             | 21159 |     0 |     53200 | 39.77 |
|   LUT as Memory            | 10784 |     0 |     17400 | 61.98 |
|     LUT as Distributed RAM | 10752 |     0 |           |       |
|     LUT as Shift Register  |    32 |     0 |           |       |
| Slice Registers            |  8044 |     0 |    106400 |  7.56 |
|   Register as Flip Flop    |  7718 |     0 |    106400 |  7.25 |
|   Register as Latch        |   326 |     0 |    106400 |  0.31 |
| F7 Muxes                   |  5394 |     0 |     26600 | 20.28 |
| F8 Muxes                   |  2688 |     0 |     13300 | 20.21 |
+----------------------------+-------+-------+-----------+-------+
```

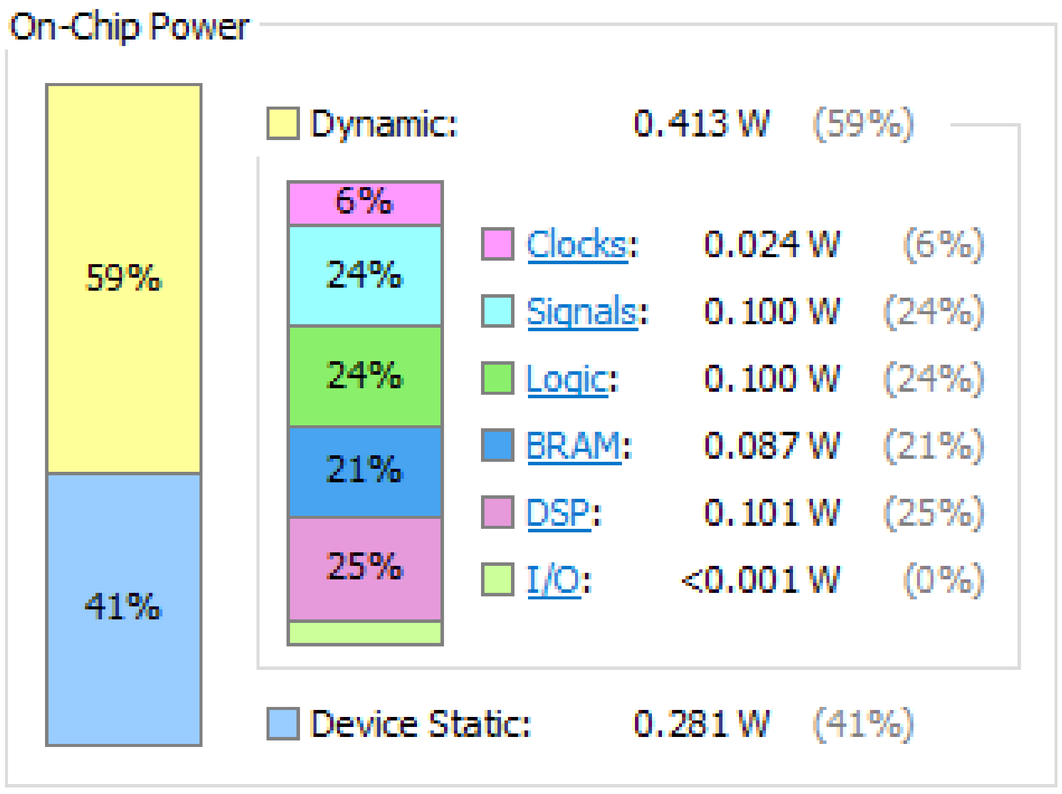
Figure 28. LUTs Utilization



```
2. Memory
---------

+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |  126 |     0 |       140 | 90.00 |
|   RAMB36/FIFO*    |  120 |     0 |       140 | 85.71 |
|     RAMB36E1 only |  120 |       |           |       |
|   RAMB18          |   12 |     0 |       280 |  4.29 |
|     RAMB18E1 only |   12 |       |           |       |
+-------------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available


3. DSP
------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           |  217 |     0 |       220 | 98.64 |
|   DSP48E1 only |  217 |       |           |       |
+----------------+------+-------+-----------+-------+
```

Figure 29. DSP and Memory Utilization

5.4. Power Consumption



On-Chip Power

Dynamic:	0.413 W	(59%)	
	Clocks:	0.024 W	(6%)
	Signals:	0.100 W	(24%)
	Logic:	0.100 W	(24%)
	BRAM:	0.087 W	(21%)
	DSP:	0.101 W	(25%)
	I/O:	<0.001 W	(0%)
Device Static:	0.281 W	(41%)	

Figure 30. Power Consumption

Chapter 6 : Conclusion & Future Work

6.1. Conclusion

In this document, the design and implementation of AlexNet have been illustrated as one of the convolutional neural network architectures on ZYNQ-702 FPGA operating on 50 MHz frequency. The target applications are robotics and portable devices, so, we focused on implementing power efficient hardware of CNN used in image classification.

We designed an architecture that fits all layers in the network, thus, the required area and resources for implementation are minimized. Memory hierarchy is used to match the dataflow. Row stationary technique is applied on 2-D convolution; hence, we could minimize data transfer rate and power consumption as much as possible. In addition to this, our design can loosely fit any network architectures not only AlexNet.

6.2. Future Work

6.2.1. Lazy Conv Pool

In this section, we propose Lazy Convolution Pooling (LCP), an approximate computational method that combines convolution and max pooling for reducing power consumption of CNN processors. We also propose Sign Connect, a simplified arithmetic unit that can be effectively used with LCP.

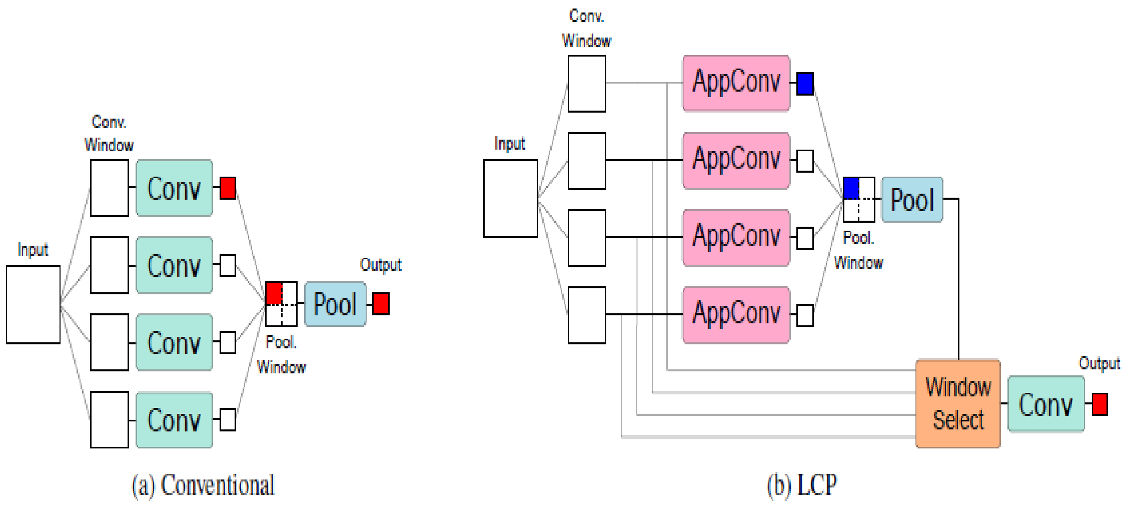


Figure 31. LCP Block

The objective of LCP is to eliminate the convolutional computation that will not affect the output of the succeeding pooling layer. Figure 2a illustrates the conventional convolution and pooling operations. In this example, four convolutions are first executed on an input image to calculate adjacent four feature values in a 2 × 2 pooling window. Then the max pooling is operated to pick up one representative feature that has a maximum value (shown as a red pixel). Worthy of attention here is that only one convolution result is passed to the next stage and all the other convolution results are discarded by the pooling operation. This means that the non-representative feature values do not affect the feature map after pooling as long as the feature selection in the pooling is correct. On the basis of this observation, we construct LCP framework with two main parts:

(1) Approximated convolution units (AppConv) to predict the feature selection in the pooling layer.

(2) A single regular (and exact) convolution unit (Conv) to perform convolution only on the predicted window. Figure 2b illustrates the LCP-based convolution and pooling operations. The computation flow of LCP is summarized as follows:

Step 1 Compute AppConv using the lightweight and approximate computation.

Step 2 Operate max pooling to the results of AppConv to specify the convolution index of which the convolution result is propagated.

Step 3 Compute the Convolution only to the input corresponding to the specified index in Step 2.

LCP can reduce the number of the exact convolutions by a factor of 1/NP, where NP is the number of the features in a pooling window. On the other hand, the prediction by App-Conv introduces additional hardware. To minimize additional power consumption associated with the prediction, we utilize lightweight and approximate computation called Sign Connect, which is described in the next subsection.

LCP executed only the necessary convolution by computing AppConv before Conv. Computational cost of AppConv must be low to realize power reduction through LCP. Here, for this purpose, we propose Sign Connect as an approximate computation unit of reduced computation complexity. Sign Connect computes product terms by the following equations:

$$\text{Sign(x)} = \begin{cases} +1\,, if\ x > 0 \\ \ 0\,, if\ x < 0 \end{cases}$$

Sign Connect can be implemented so that general multipliers are replaced by the multiplications of the sign of weights, which can be typically realized by multiplexers. With Sign Connect, multiplication could be drastically simplified, and the total performance becomes deterministic.

6.2.2. Dual-Port Caches

To speed up the operation of layers 2-5, a valid solution is to make the caches two port and double its size, this will be applicable on larger boards such as Virtex-7. Smaller boards will have the problem of insufficient block RAMs. Generally, any dual-port memory will be synthesized as a block RAM, so implementing it as a LUT as memory will not be possible. However this step will drastically increase the power almost the double, so this solution wasn't considered from the start as the target is low power.

Layer	Number of channels before modification	Number of channels after modification
1	8	12
2	16	32
3	16	32
4	16	32

6.2.3. Clock Gating

An important aspect when it comes to reducing power consumption is to stop all the blocks which aren't active at that time. To do so, clock gating is used to stop any switching activity in those blocks. Mainly, clock gating is implemented using a Flip-Flop and an AND gate. This reduces the probability of glitches and metastability, as the only delay between the enable signal and the clock signal will be the setup delay of the AND gate. The clock gating circuit is shown in fig.().

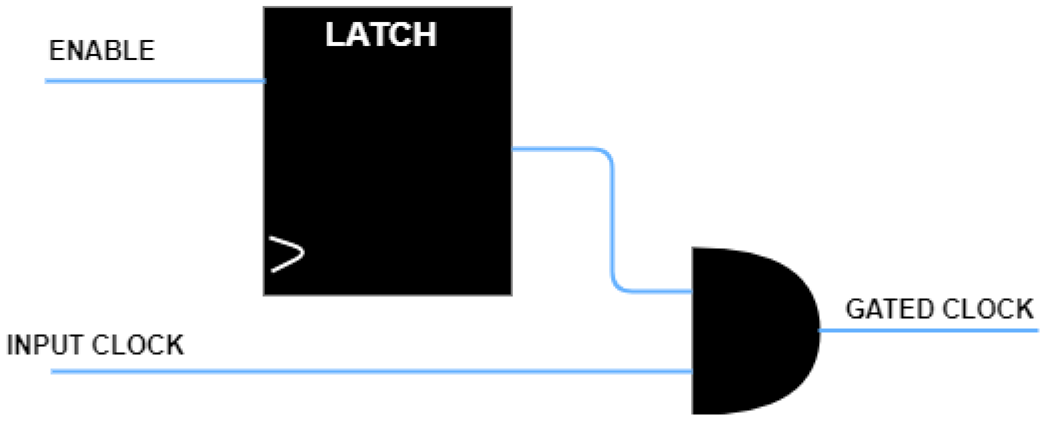


Figure 32. Gated Clock

References

1. Efficient Processing of Deep Neural Networks: A Tutorial and Survey , By Vi v i en n e Sz e , Senior Member IEEE, Yu-Hs in Chen, Student Membe r IEEE, Tien-Ju Yang, Student Member IEEE, and Joel S. Emer, Fellow IEEE.Newmark , N.M ., and Resenblueth E., 1971, Fundamentals of Earthquake Engineering, Vol. xx, 2nd edition, Prentice – Hall Inc ., Englewood cliffs , N.J.
2. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, no. 7553, pp. 436–444, May 2015.
3. L. Deng, et al., "Recent advances in deep learning for speech research at Microsoft," in Proc. ICASSP, 2013, pp. 8604–8608.INFORMS web site, January 2012, http://www.informs.org.
4. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Proc. NIPS, 2012, pp. 1097–1105.
5. Method to Estimate the Energy Consumption of Deep Neural Networks , Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, Vivienne Sze Massachusetts Institute of Technology, Cambridge, MA, USA ftjy, yhchen, jsemer, szeg@mit.edu
6. Approximated Prediction Strategy for Reducing Power Consumption of Convolutional Neural Network Processor Takayuki Ujiie Masayuki Hiromoto Takashi Sato Graduate School of Informatics, Kyoto University Yoshida-hon-machi, Sakyo, Kyoto, Japan paper@easter.kuee.kyoto-u.ac.j