Self Driving Car: Steering angle prediction through FPGA accelerated Deep Learning

17.01.2016

Team Members

Zain Kabir	05120
Osama Waqar	06179
Saad Qureshi	05804
Usman Mahmood	05373
Abdullah Ahsan	05396
Uzair Akbar	04584

Contents

List of Figures

1 Overview

This is in partial fulfillment of a embedded systems design course semester project. In short, the project predicts self-driving car's steering angle through deep learning (Convolutional Neural Network) on FPGA with increased throughput and enhanced latency.

2 Motivation

Till this date, more than 1.25 million casualties have been reported in road traffic accidents[1].Most of these fatalities caused due to carelessness of driver. Many safety measures have been ensured over the time but death tolls are still rising due to reckless driving[2]. An everlasting solution to this fatalities could be autonomous cars that navigate through the streets/roads independently, hence eliminating the possibility of any human error. Imagine sitting in one of these cars having no fear of accident, spending your time with family while car drives itself autonomously for you.

These autonomous vehicles are very safe. Stats from "Google's self-driving car" reveal only 12 accident over period of 12 months, each of them caused due to negligence of other drivers on the road.

Algorithms for self-driving car's seamless navigation are designed day in and day out. This area is still research-in-progress and has many challenges associated with it. One of the active dimension currently is to predict steering angle of self-driving car expeditiously to make self-driving car experience more safe and real-time.

3 Project Rationale

Most indispensable part of self-driving car is predicting steering angle. Till now, many machine learning algorithms have been deployed to predict steering angle. Due to limitation of Processors used and complexity of Machine Learning algorithms, a trade off always exist between latency and accuracy of model.

Advanced deep learning algorithms though provide accurate results but require high number of computation depreciating latency and hence limiting real-time effect of system. On the other hand deep learning algorithms give highly accurate results.

A solution to this problem is to implement Deep learning Algorithms on FPGA. Realizing the parallel operation we can execute numerous computations simultaneously, thus-by achieving precise output with improved accuracy and enhanced latency.

4 Current Work

Google Self driving car WAYMO project currently utilizes LIDAR technology[3].This car generates the 3D map of environment by illuminating target with Laser Light. The most expensive part of this technology is LIDAR system. It is designed to detect pedestrians, cyclists, vehicles, road work and more from a distance of up to two football fields away in all directions.

Tesla Autopilot is another autonomous vehicle that visualizes feed from Eight surround cameras providing it with 360 degrees of visibility up to 250 meters of range[4].Twelve updated ultrasonic sensors complement this vision, allowing for detection of both hard and soft objects at nearly twice the distance of the prior system.

Udacity's self-driving is another leap forward where car's vision Primarily depend upon camera feed. Deep Learning algorithms are applied on these feeds to find road signs, vehicles and steer through the roads autonomously[5].Challenge now exists to improve latency of deep learning algorithms ensuring enhanced efficiency.

5 Our Solution

To advance the research-in-progress for predicting car's steering angle primarily by using camera feed, we are building an embedded system that accurately predicts the steering angle ensuring improved efficiency.

Deep learning algorithm such as convolutional neural network that are optimized for providing accurate output in such scenarios are used. We aim to train our model on Tensor Flow using pre-existing dataset.

This model would then be implemented on FPGA. Realizing plenty of Adders and Multipliers on FPGA we can achieve parallel processing on input image thus by ensuruing significant drop in latency and improvement in throughput of CNN.

6 Convolutional Neural Network

Convolutional neural network (CNN/ConvNets) is a Machine Learning technique that is inspired after brain structure. CNN is an advanced form of Neural Network which involves complex operations on neurons such as 4D-convolution in multiple layers, pooling, relu and FCL operation before producing the output. The extensive amount of computation on every neuron result in highly accurate output of this network.

CNN nowadays is popular choice for visual recognition tasks. Input to the ConvNets are Image pixels.

A typical model of ConvNets is shown below

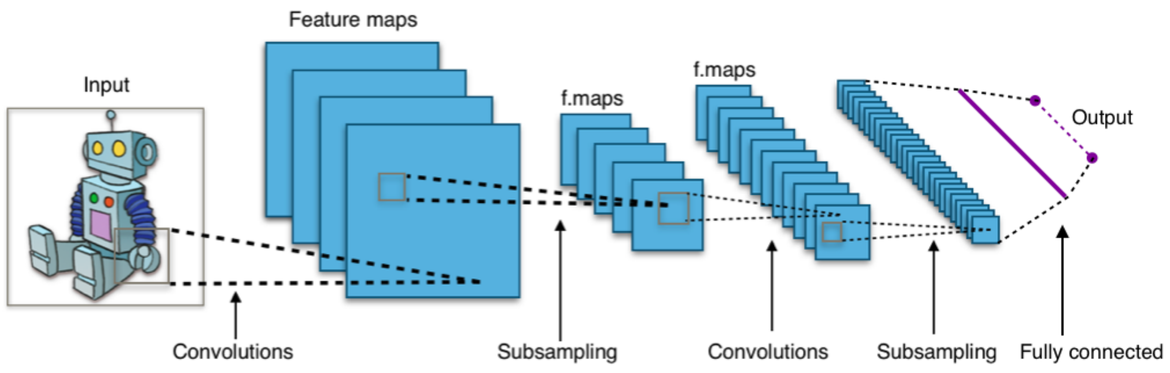


Figure 1: Typical convolutional neural network architecture.

6.1 Convolution

The fundamental and most essential part of CNN is convolution. Input image pixels are convolved with already trained kernels on a given data set. Convolution type varies from model to model. In typical models convolution could vary from 2D, 3D to 4D. More the dimensions higher the complexity of model better the model accuracy.

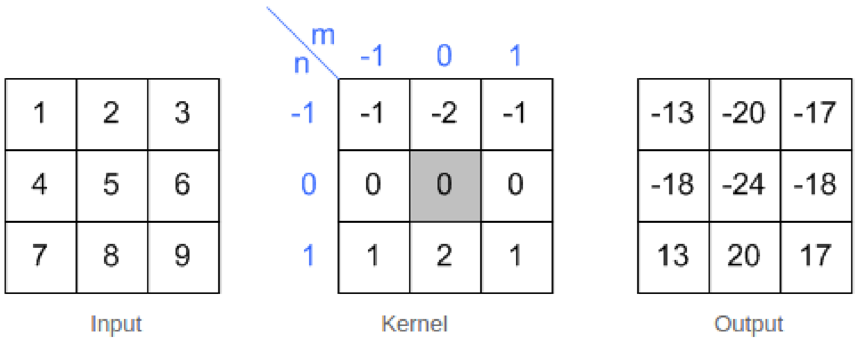


Figure 2: A typical 2D convolution is shown here where a stride (number of pixels by which we slide our kernel on Input matrix at each stage) of 1 is used.

6.2 Supsampling/Pooling

A typical 2D convolution is shown here where a stride (number of pixels by which we slide our kernel on Input matrix at each stage) of 1 is used.

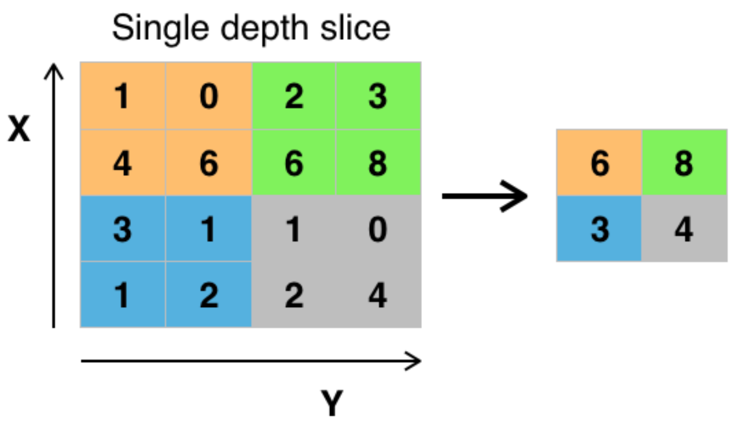


Figure 3: Example of Max Pooling where 4x4 matrix is downsampled to 2x2 by picking maximum value in corresponding 2x2 block.

There are several ways to downsample.

- Max pooling
- Average pooling
- L2 norm pooling

Other than this some customized way of downsampling could also be defined.

6.3 Relu Operation

Rectified Linear Units (Relu) is another approach in CNNs to reduce the computational complexity. It increases the nonlinear properties of CNN without almost negligible impact on accuracy of network.

Use of ReLU operator is preferable, since it results in the neural network training several times faster, without making a significant difference to generalisation accuracy.

7 Why Convolutional Neural Networks?

Convolutional Neural Networks are one of the best tools used for image recognition applications. Some examples of their application include the highest ever achieved accuracy rates on the MINST database[] and the fastest learning rates (for the same accuracy rating) in cases like the NORB database[]. CNNs hence provide a good trade-off between good accuracy rates and learning rates in image classification problems. The same advantages translate over to regression problems based on images. However, one of the disadvantages of the convolutional neural network is very low inference rates due to the expensive convolution operations in the network. Therefore, if the convolution operation is implemented in parallel (which is what the whole network boils down to) can have a significant improvement on the overall inference rates of the network. If implemented properly, our parallel implementation satisfy all three requirements of high accuracy rate, high learning rate and high inference rates.

8 Efficient Implementation of Conv Nets

Since CNN architecture involves multiple operations it becomes difficult for a processor to perform such computation. To make the processor's output real-time efficient implementations of CNN are realized. A brief overview of both traditional and efficient implementation of CNN are discussed below from computational complexity perspective.

8.1 Traditional Approach

Traditional approach used for calculating output of convolution requires sliding kernel over the input matrix. Multiplying the overlapping matrix indices gives only one index of output matrix. Sliding of kernel over the input matrix is done again and next index of output matrix is received. Process is repeated until we have all the indices of output matrix.

A traditional way implementing 2D-convolution is graphically demonstrated here.

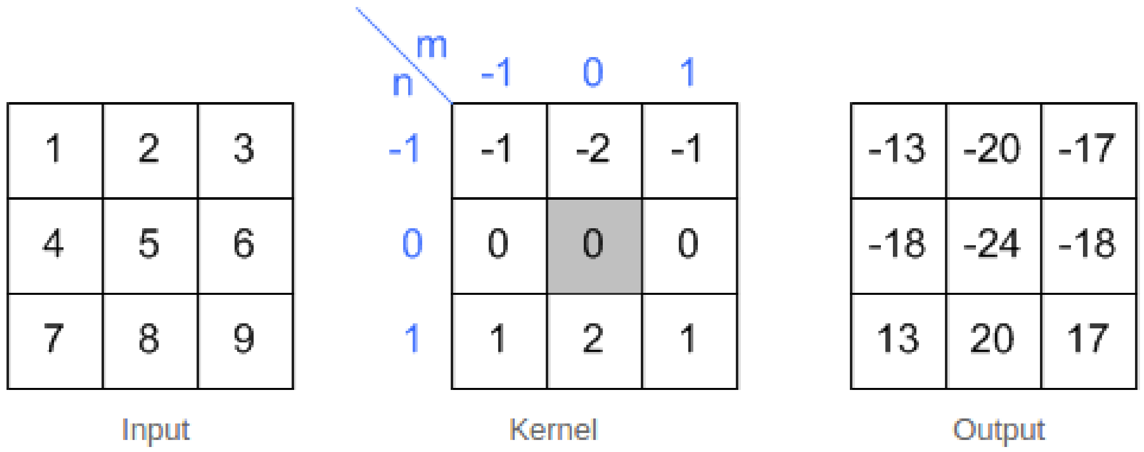
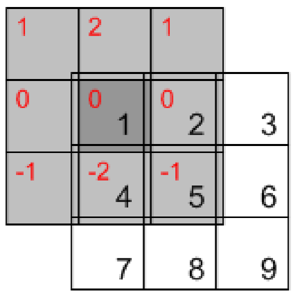


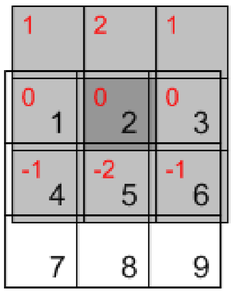
Figure 4: Traditional kernel

For first index [0,0] of output matrix following computation is done.



$$\begin{aligned} y[0,0] &= x[-1,-1]\cdot h[1,1] + x[0,-1]\cdot h[0,1] + x[1,-1]\cdot h[-1,1] \\ &+ x[-1,0]\cdot h[1,0] + x[0,0]\cdot h[0,0] + x[1,0]\cdot h[-1,0] \\ &+ x[-1,1]\cdot h[1,-1] + x[0,1]\cdot h[0,-1] + x[1,1]\cdot h[-1,-1] \\ &= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 0\cdot 0 + 1\cdot 0 + 2\cdot 0 + 0\cdot(-1) + 4\cdot(-2) + 5\cdot(-1) = -13 \end{aligned}$$

For first index [1,0] of output matrix following computation is done.



$$\begin{aligned} y[1,0] &= x[0,-1]\cdot h[1,1] + x[1,-1]\cdot h[0,1] + x[2,-1]\cdot h[-1,1] \\ &+ x[0,0]\cdot h[1,0] + x[1,0]\cdot h[0,0] + x[2,0]\cdot h[-1,0] \\ &+ x[0,1]\cdot h[1,-1] + x[1,1]\cdot h[0,-1] + x[2,1]\cdot h[-1,-1] \\ &= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 1\cdot 0 + 2\cdot 0 + 3\cdot 0 + 4\cdot(-1) + 5\cdot(-2) + 6\cdot(-1) = -20 \end{aligned}$$

And so on for rest of output matrix indices.

8.2 Computation Complexity

In traditional ConvNet implementation computational complexity is:

$$O(m \times n \times k \times k)$$

where (m,n) are dimensions of Input matrix, whereas (k,k) is the dimension of kernel matrix.

8.3 Modified Implemenation

An efficient implementation of 2D-convolutions that significantly reduces the computational complexity is presented. This approach involves applying MAC operation and shift operation on every pixel. Interesting outcome of this implementation could be viewed that at every clock cycle an index of output matrix is received.

Model working is graphically illustrated in figure below,

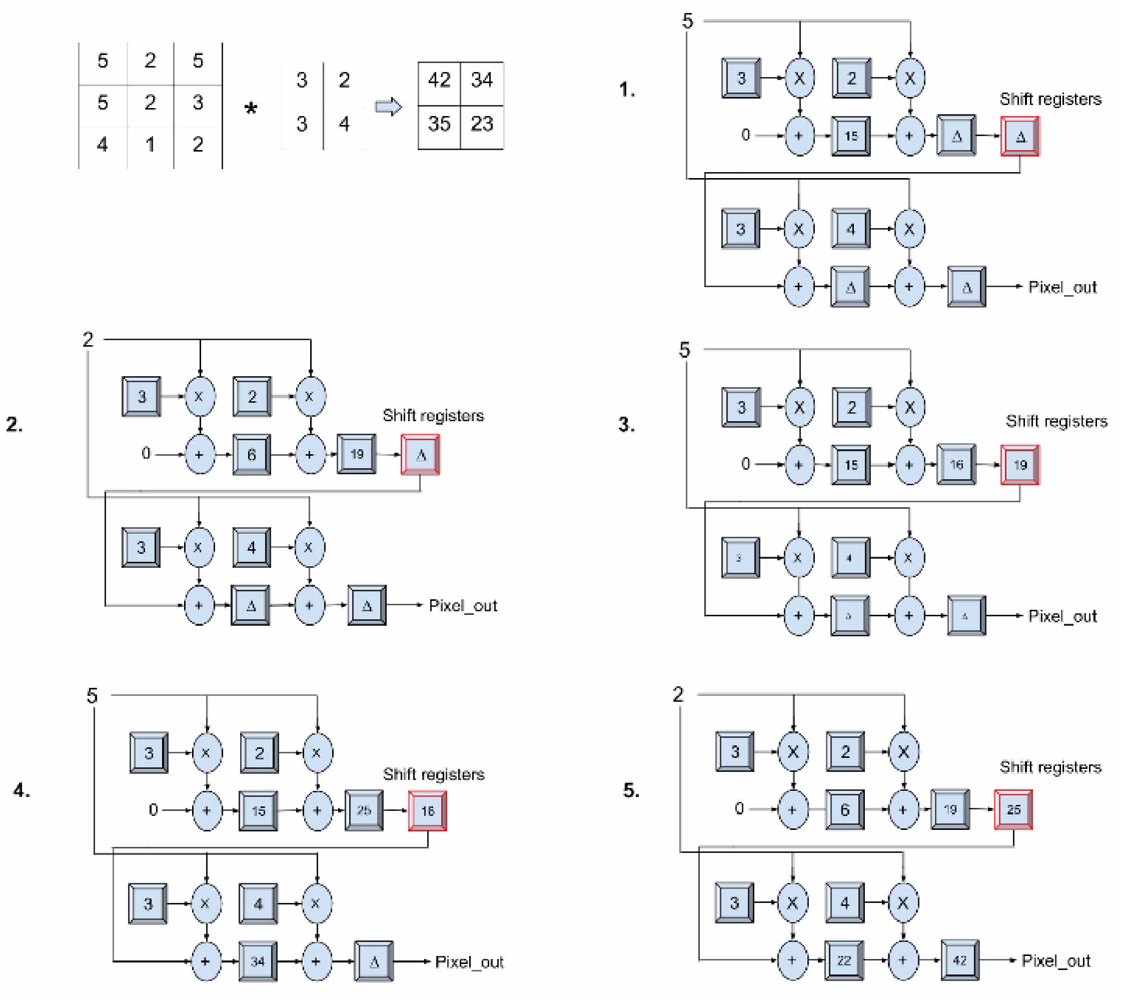


Figure 5: Working model of implementation

8.4 Computation Complexity

In efficiently implemented ConvNet computational complexity is reduced to:

$$O(m \times n)$$

where (m,n) are dimensions of Input matrix.O

9 Our CNN Model

9.1 1st Layer of 4D-Convolution

Since most extensive computational power of ConvNets is required in 1st layer of CNN hence it is implemented in FPGA to reduce the latency of our model.

Input of Layer-1 is 3D matrix of dimension 200x66x3

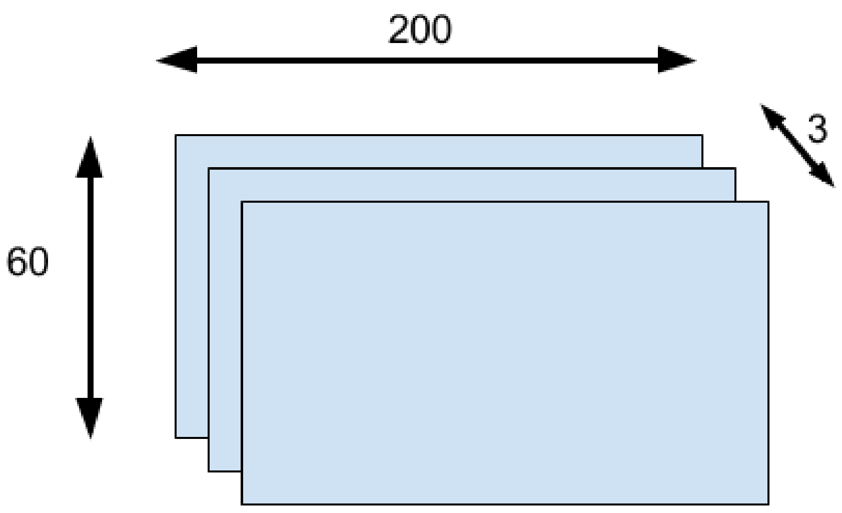


Figure 6: Image input with pixel dimension

Where 200x66 specify dimension of pixels in image. 3 represent RGB (Red, Green, Blue) channels of Input Image.

This input matrix get convolved trained kernels of following dimensions: 3x3x3x6

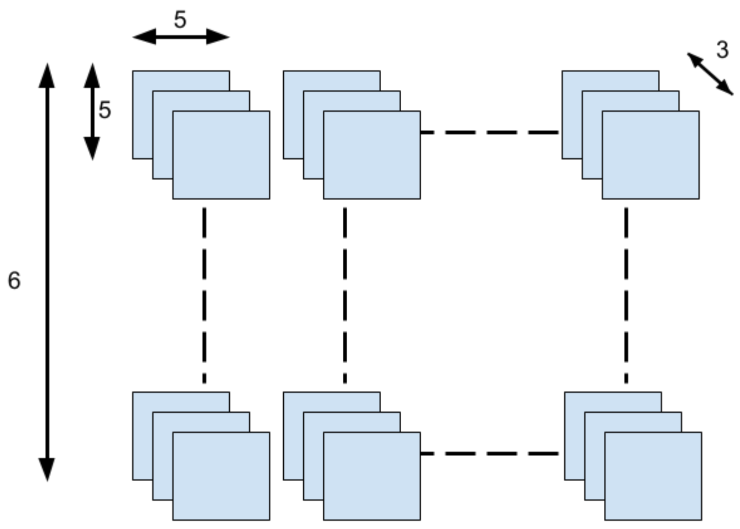


Figure 7: Image input with pixel dimension

where, 3x3 are dimensions of image, 3 being RGB kernels and 6 is depth of kernels.

9.2 2nd Layer of 4D-Convolution

Output of 1st Layer of Convolution is a 32x99x24 dimension matrix. Now further layer of ConvNets are implemented on PC. input to 2nd layer of convolution becomes 32x99x24 that gets convolved with kernels of dimension 3x3x6x8.

In all layers of convolution stride of 2 is ensured.

9.3 3rd Layer of 4D-Convolution

Output of 2nd Layer of Convolution is a 15x49 dimension matrix.

Input to 3rd layer of convolution becomes 15x49 that gets convolved with kernels of dimension 3x3x8x8.

9.4 4th Layer of 4D-Convolution

Output of 3rd Layer of Convolution is a 7x24 dimension matrix. Input to 3rd layer of convolution becomes 7x24 that gets convolved with kernels of dimension 3x3x8x10.

10 Schematic Diagram

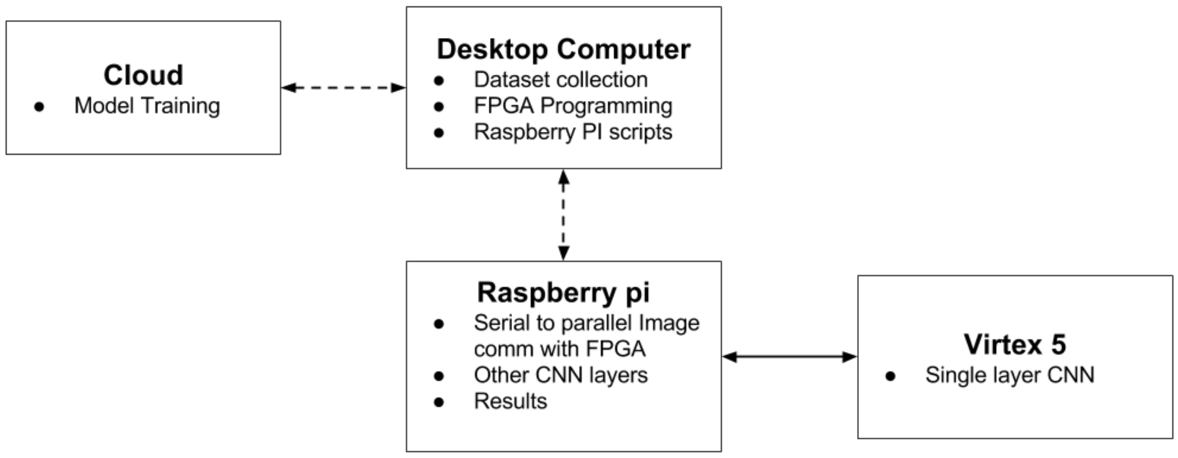


Figure 8: Block diagram

10.1 Cloud

The training of the convolutional neural network model was done on remote PCs with substantial GPU power. Microsoft Azure virtual machine instances were used to run the TensorFlow scripts on the training data. As the training data is in tens of GBs, its training demanded the use of these powerful cloud computers.

10.2 Desktop Computer

A desktop computer is used to access the image data set and the trained model from the cloud instances. This desktop computer is responsible for pre-processing the images and then copying the data set over to the raspberry pi. It is used to program the FPGA and make the python scripts for the Raspberry PI.

10.3 Raspberry PI

The role of the raspberry pi is to send pre-processed images to the FPGA. It also implements the remaining layer of the CNN. It takes the image dataset input from the PC. It communicates this data to the FPGA in a parallel fashion by connecting to its pins with the GPIO. There are 8 data pins in this custom bus and one clock pin. FPGA returns kernel parameters to the Pi which are then used for the remaining CNN layers. The output angle is then checked against the actual steering angles in the images.

10.4 Virtex 5 FPGA

Virtex 5 on the ML507 evaluation board is the main heavy lifter of this whole system. It performs the convolutional neural network operations on the input values and computes the next level kernel parameters. Specifically, it implements the first layer of the CNN directly in the logic blocks. It computes the output and then sends it to the raspberry pi for further processing.

11 Control/Flow Diagram

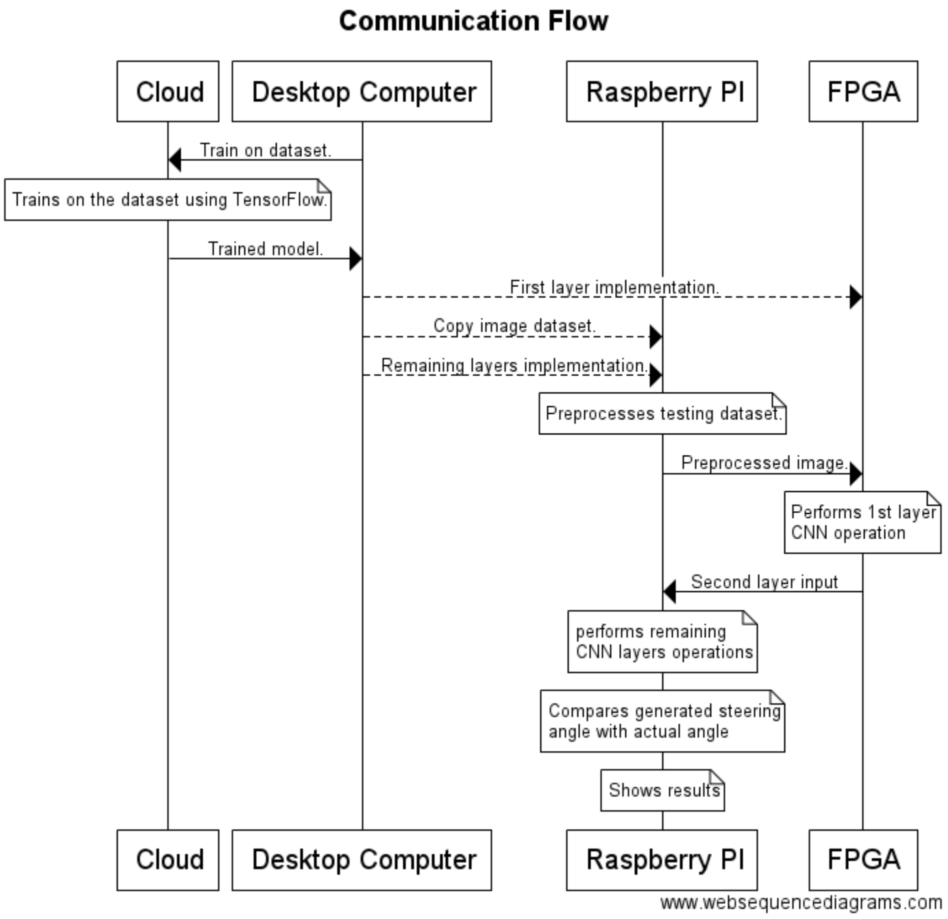

Communication Flow
Cloud
Desktop Computer
Raspberry PI
FPGA
Train on dataset.
Trains on the dataset using TensorFlow.
Trained model.
First layer implementation.
Copy image dataset.
Remaining layers implementation.
Preprocesses testing dataset.
Preprocessed image.
Performs 1st layer CNN operation
Second layer input
performs remaining CNN layers operations
Compares generated steering angle with actual angle
Shows results
Cloud
Desktop Computer
Raspberry PI
FPGA
www.websequencediagrams.com

Figure 9: Communication/Flow Diagram

11.0.1 Cloud

The training of the convolutional neural network model was done on remote PCs with substantial GPU power. Microsoft Azure virtual machine instances were used to run the TensorFlow scripts on the training data. As the training data is in tens of GBs, its training demanded the use of these powerful cloud computers.

11.0.2 Desktop Computer

A desktop computer is used to access the image data set and the trained model from the cloud instances. This desktop computer is responsible for pre-processing the images and then copying the data set over to the raspberry pi. It is used to program the FPGA and make the python scripts for the Raspberry PI.

11.0.3 Raspberry PI

The role of the raspberry pi is to send pre-processed images to the FPGA. It also implements the remaining layer of the CNN. It takes the image dataset input from the PC. It communicates this data to the FPGA in a parallel fashion by connecting to its pins with the GPIO. There are 8 data pins in this custom bus and one clock pin. FPGA returns kernel parameters to the Pi which are then used for the remaining CNN layers. The output angle is then checked against the actual steering angles in the images.

11.0.4 Virtex 5

Virtex 5 on the ML507 evaluation board is the main heavy lifter of this whole system. It performs the convolutional neural network operations on the input values and computes the next level kernel parameters. Specically, it implements the rst layer of the CNN directly in the logic blocks. It computes the output and then sends it to the raspberry pi for further processing.

12 Hardware Components

The following hardware components are used in our project:

- Cloud VM (2× K80 NVIDIA Tesla, 24 GB Video Memory)
- Desktop PC (for raspberry pi scripts development and FPGA programming)
- Raspberry Pi 3
- Virtex 5 FPGA on ML507 Board

13 Software/Libraries Used

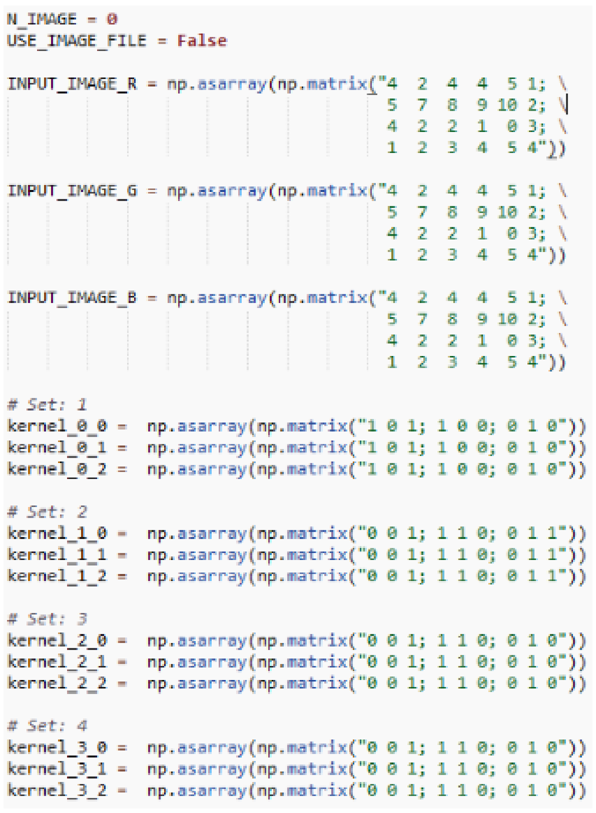
- SciPy
- TensorFlow

14 User Manual

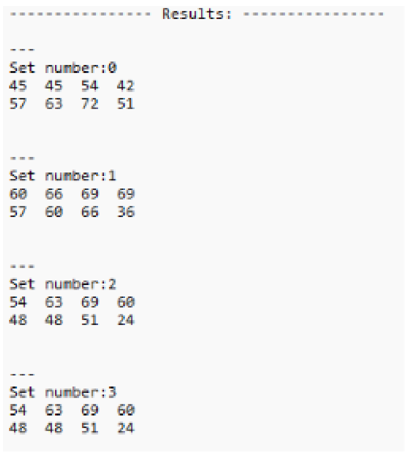
- Connect the hardware as shown in this document
- A parallel bus should be present between the Raspberry PI and the FPGA as shown before in this document
- Load the verilog program and download it into the FPGA

- Identify the image source (video stream/ camera etc)
- Obtain the image data stream in python
- Pass the image stream as a parameter to the python scrupt
- The python script will then produce the output angle and show the results

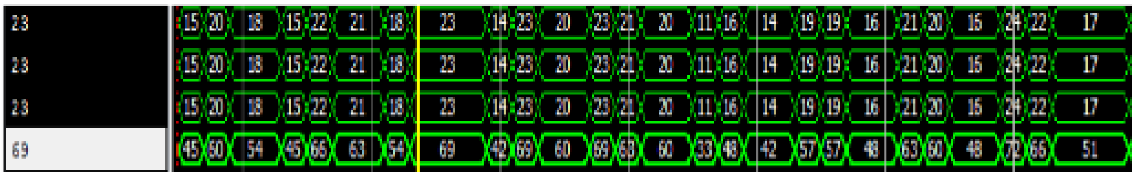
15 Snapshots



```
N_IMAGE = 0
USE_IMAGE_FILE = False

INPUT_IMAGE_R = np.asarray(np.matrix("4  2  4  4  5 1; \
                                      5  7  8  9 10 2; \
                                      4  2  2  1  0 3; \
                                      1  2  3  4  5 4"))

INPUT_IMAGE_G = np.asarray(np.matrix("4  2  4  4  5 1; \
                                      5  7  8  9 10 2; \
                                      4  2  2  1  0 3; \
                                      1  2  3  4  5 4"))

INPUT_IMAGE_B = np.asarray(np.matrix("4  2  4  4  5 1; \
                                      5  7  8  9 10 2; \
                                      4  2  2  1  0 3; \
                                      1  2  3  4  5 4"))

# Set: 1
kernel_0_0 =  np.asarray(np.matrix("1 0 1; 1 0 0; 0 1 0"))
kernel_0_1 =  np.asarray(np.matrix("1 0 1; 1 0 0; 0 1 0"))
kernel_0_2 =  np.asarray(np.matrix("1 0 1; 1 0 0; 0 1 0"))

# Set: 2
kernel_1_0 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 1"))
kernel_1_1 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 1"))
kernel_1_2 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 1"))

# Set: 3
kernel_2_0 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 0"))
kernel_2_1 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 0"))
kernel_2_2 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 0"))

# Set: 4
kernel_3_0 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 0"))
kernel_3_1 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 0"))
kernel_3_2 =  np.asarray(np.matrix("0 0 1; 1 1 0; 0 1 0"))
```



```
---------------- Results: ----------------

---
Set number:0
45  45  54  42
57  63  72  51


---
Set number:1
60  66  69  69
57  60  66  36


---
Set number:2
54  63  69  60
48  48  51  24


---
Set number:3
54  63  69  60
48  48  51  24
```



16 Complete Code

16.1 CNN Training Python Script

16.1.1 CNN Model Block

```
import tensorflow as tf
import scipy

def weight_variable(shape):
initial = tf.truncated_normal(shape, stddev=0.1)
return tf.Variable(initial)

def bias_variable(shape):
initial = tf.constant(0.1, shape=shape)
return tf.Variable(initial)

def conv2d(x, W, stride):
return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding='
    ↪ VALID')

x = tf.placeholder(tf.float32, shape=[None, 66, 200, 3])
y_ = tf.placeholder(tf.float32, shape=[None, 1])

x_image = x

#first convolutional layer
W_conv1 = weight_variable([3, 3, 3, 6])
b_conv1 = bias_variable([6])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1, 2) + b_conv1)
print(h_conv1.get_shape())

#second convolutional layer
W_conv2 = weight_variable([3, 3, 6, 8])
b_conv2 = bias_variable([8])

h_conv2 = tf.nn.relu(conv2d(h_conv1, W_conv2, 2) + b_conv2)
print(h_conv2.get_shape())

#third convolutional layer
W_conv3 = weight_variable([3, 3, 8, 8])
b_conv3 = bias_variable([8])

h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 2) + b_conv3)
print(h_conv3.get_shape())

#fourth convolutional layer
W_conv4 = weight_variable([3, 3, 8, 10])
b_conv4 = bias_variable([10])

h_conv4 = tf.nn.relu(conv2d(h_conv3, W_conv4, 2) + b_conv4)
print(h_conv4.get_shape())

#fifth convolutional layer
```

```
W_conv5 = weight_variable([3, 3, 10, 10])
b_conv5 = bias_variable([10])

h_conv5 = tf.nn.relu(conv2d(h_conv4, W_conv5, 1) + b_conv5)
print(h_conv5.get_shape())

#FCL_1
W_fc1 = weight_variable([9000, 1164])
b_fc1 = bias_variable([1164])

h_conv5_flat = tf.reshape(h_conv5, [-1, 9000])
h_fc1 = tf.nn.relu(tf.matmul(h_conv5_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
print(h_fc1.get_shape())

#FCL_2
W_fc2 = weight_variable([1164, 100])
b_fc2 = bias_variable([100])

h_fc2 = tf.nn.relu(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)
print(h_fc2.get_shape())

#FCL_3
W_fc3 = weight_variable([100, 50])
b_fc3 = bias_variable([50])

h_fc3 = tf.nn.relu(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)

h_fc3_drop = tf.nn.dropout(h_fc3, keep_prob)
print(h_fc3.get_shape())

#FCL_4
W_fc4 = weight_variable([50, 10])
b_fc4 = bias_variable([10])

h_fc4 = tf.nn.relu(tf.matmul(h_fc3_drop, W_fc4) + b_fc4)

h_fc4_drop = tf.nn.dropout(h_fc4, keep_prob)
print(h_fc4.get_shape())

#Output
W_fc5 = weight_variable([10, 1])
b_fc5 = bias_variable([1])


y = tf.mul(tf.atan(tf.matmul(h_fc4_drop, W_fc5) + b_fc5), 2) #scale
    ↪ the atan output
print(y.get_shape())
```

16.1.2 TensorFlow Training Block

```
import os
import tensorflow as tf
import driving_data
import model
import numpy as np

LOGDIR = './save'

sess = tf.InteractiveSession()

L2NormConst = 0.001

train_vars = tf.trainable_variables()

loss = tf.reduce_mean(tf.square(tf.sub(model.y_, model.y))) + tf.
    ↪ add_n([tf.nn.l2_loss(v) for v in train_vars]) * L2NormConst
train_step = tf.train.AdamOptimizer(1e-4).minimize(loss)
sess.run(tf.initialize_all_variables())

# create a summary to monitor cost tensor
tf.scalar_summary("loss", loss)
# merge all summaries into a single op
merged_summary_op = tf.merge_all_summaries()

saver = tf.train.Saver()

#op_to_write_logs_to_Tensorboard
logs_path = './logs'
summary_writer = tf.train.SummaryWriter(logs_path, graph=tf.
    ↪ get_default_graph())

epochs = 30
batch_size = 100

np.set_printoptions(threshold=np.nan)

#train_over_the_dataset_about_30_times
for epoch in range(epochs):
for i in range(int(driving_data.num_images/batch_size)):
xs, ys = driving_data.LoadTrainBatch(batch_size)
train_step.run(feed_dict={model.x: xs, model.y_: ys, model.
    ↪ keep_prob: 0.8})
if i % 10 == 0:
xs, ys = driving_data.LoadValBatch(batch_size)
loss_value = loss.eval(feed_dict={model.x:xs, model.y_: ys, model.
    ↪ keep_prob: 1.0})
print("Epoch: %d, Step: %d, Loss: %g" % (epoch, epoch * batch_size
    ↪ + i, loss_value))

#write_logs_at_every_iteration
```

```
summary = merged_summary_op.eval(feed_dict={model.x:xs, model.y_:
    ↪ ys, model.keep_prob: 1.0})
summary_writer.add_summary(summary, epoch * batch_size + i)

if i % batch_size == 0:
if not os.path.exists(LOGDIR):
os.makedirs(LOGDIR)
checkpoint_path = os.path.join(LOGDIR, "model.ckpt")
filename = saver.save(sess, checkpoint_path)
print("Model saved in file: %s" % filename)

if epoch >= 29:
all_vars = tf.trainable_variables()
for v in all_vars:
direc = 'trained_kernels/'
f = open(direc + v.name[:-2] + '.txt', 'w+')
f.write(str(v.eval()))
f.close()
print("Run the command line:\n" \
"--> tensorboard --logdir=./logs " \
"\nThen open http://0.0.0.0:6006/ into your web browser")
```

16.2 Verilog

16.2.1 MAC Block

```
`timescale 1ns / 1ps
////////////////////////////////////////
// Multiplies and accumulates result in one clock cycle
////////////////////////////////////////
module mac(
input [7:0] in,
input [7:0] w,
input [7:0] b,
output [15:0] out
);
wire [15:0] d;
assign d = w * in;
assign out = d + b;

endmodule
```

16.2.2 Conv Layer 1

```
`timescale 1ns / 1ps

module conv_l1(
input clk,
input reset,
input [7:0] pxl_in,
// 3*3 kernel values
input [7:0] kernel_00, input [7:0] kernel_01, input [7:0] kernel_02
    ↪ ,
input [7:0] kernel_10, input [7:0] kernel_11, input [7:0] kernel_12
    ↪ ,
input [7:0] kernel_20, input [7:0] kernel_21, input [7:0] kernel_22
    ↪ ,

// Outputs against each MAC register and a shift register after
    ↪ each row
output [15:0] rline00, output [15:0] rline01, output [15:0]
    ↪ rline02, output [15:0] rlines1,
output [15:0] rline10, output [15:0] rline11, output [15:0] rline12
    ↪ ,                      output [15:0] rlines2,
output [15:0] rline20, output [15:0] rline21, output [15:0] rline22
    ↪ ,                      output [15:0] pxl_out,
output valid
);
wire [15:0] wire00; wire [15:0] wire01; wire [15:0] wire02;
wire [15:0] wire10; wire [15:0] wire11; wire [15:0] wire12;
wire [15:0] wire20; wire [15:0] wire21; wire [15:0] wire22;

// Row : 1

mac mac00(pxl_in, kernel_00, 0, wire00);
shift_mac_l1 sm00(clk, wire00, rline00);

mac mac01(pxl_in, kernel_01, rline00, wire01);
shift_mac_l1 sm01(clk, wire01, rline01);

mac mac02(pxl_in, kernel_02, rline01, wire02);
shift_mac_l1 sm02(clk, wire02, rline02);
shift_row_l1 sr1(clk, rline02, rlines1);

// Row : 2

mac mac10(pxl_in, kernel_10, rlines1, wire10);
shift_mac_l1 sm10(clk, wire10, rline10);

mac mac11(pxl_in, kernel_11, rline10, wire11);
shift_mac_l1 sm11(clk, wire11, rline11);

mac mac12(pxl_in, kernel_12, rline11, wire12);
shift_mac_l1 sm12(clk, wire12, rline12);
shift_row_l1 sr2(clk, rline12, rlines2);

// Row : 3
mac mac20(pxl_in, kernel_20, rlines2, wire20);
shift_mac_l1 sm20(clk, wire20, rline20);

mac mac21(pxl_in, kernel_21, rline20, wire21);
shift_mac_l1 sm21(clk, wire21, rline21);
```

```
mac mac22(pxl_in, kernel_22, rline21, wire22);

assign pxl_out = wire22;//rline22;

reg [8:0] counter1;
reg [4:0] counter2;
reg temp;

always @(posedge clk) begin
if (counter2 < 2) begin
counter2<=0;
temp<=0;
counter1<=65; end
else if (counter1 < 65) begin
counter1 <= counter1 + 1;
temp <= 0; end
else if (counter1 < 77) begin
counter1 <= counter1 + 1;
temp <= 1; end
else if (counter1 < 90) begin
counter1 <= counter1 + 1;
temp <= 0; end
else if (counter1 > 89) begin
temp<=0;
counter1<=65;
counter2<=counter2+1; end
else begin
temp <= 1;
counter1 <= counter1 + 1;
counter2 <= counter2 + 1;
counter1 <= 0;
counter2 <= 0;
end
end
assign valid = temp;

endmodule
```

17 References

[1]yann.lecun.com/exdb/publis/pdf/farabet-iscas-10.pdf
[2]https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115
[3]https://waymo.com/tech/
[4]https://www.tesla.com/autopilot
[5]https://www.udacity.com/drive
[6]http://cs231n.github.io/convolutional-networks/

[1]http://ieeexplore.ieee.org/document/5272559/

[3]http://www.who.int/violence_injury_prevention/road_safety_status/2015/en/

[5]http://github.com/24-hours/sdc
[6]http://www.songho.ca/dsp/convolution/convolution2d_example.html