2020

Digit Classification using DNNWEAVER2

SIMULATION OF LENET DIGIT CLASSIFIER IN DNNWEAVER2

SHWETA UDGIRI
SUMANISH SARKAR
JAYAKUMAR SUNDARAM

10-MARCH-2020

Contents

1. Introduction

This document describes the caffe training of LeNet model , data generation process of images and weights for open source DNNWeaver2 CNN accelerator and Digits classification simulation in DNNWeaver2 CNN accelerator.

2. LeNet Architecture

The LeNet architecture consists of two sets of convolutions and pooling layers, two sets of fully connected layers and finally a SoftMax classifier.

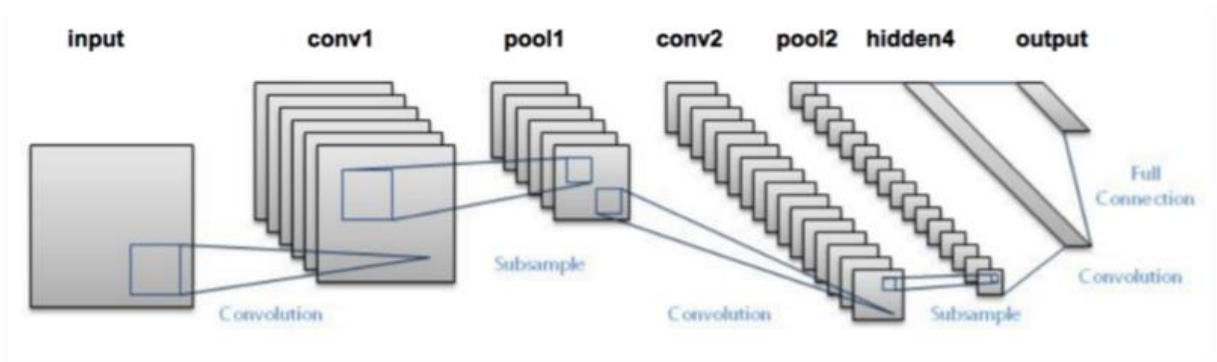

Figure 1: LeNet Architecture

As shown in Figure1, the two convolution layers are conv1, conv2, sub-sampling or poolinglayers are pool1, pool2. The Kernel size for each convolutions layers is 5x5 with stride 1 and maxpool kernel size is 2x2 with stride 2. There is a hidden fully connected layer (FC1), followed by the output layer (FC2).

Conv1 layer generates 20 output feature maps and Con2 layer generates 50 output channels. The number of neurons in FC1 layer is 500 and the output FC2 layers contains 10 (20-50-500-10). The input is a grey scale image of size 28x28.

LeNet architecture is widely used for Handwritten Digits Classifications.

The LeNet prototxt model is available in the below path

>**/FPGACNNProject/dnnWeaver2/dataGeneration/prototxt/lenet/lenet.prototxt**

3. MNIST Dataset for Training

The LeNet model is trained using the MNIST Digits dataset. The dataset contains 6000 handwritten digits 0-9 of size 28x28 for training and another 3000 data for testing.

Sample MNIST dataset look like below,

Figure 2: Sample MNIST Data

The MNIST dataset is available at
dnnWeaver2/dataGeneration/dataset/mnist

4. LeNet Model Training using caffe

This section describes the steps to train the LeNet model in Caffe framework.
Requirements:

- Caffe tools framework
- Python 3.6

The prototxt used for caffe framework training is available at
dnnWeaver2/dataGeneration/prototxt/lenet/ lenet_train_test.prototxt.

Source the Caffe environment setup script **dnnWeaver2/dataGeneration/env/rccaffe**

Run the below script to train the model using MNIST dataset.
dnnWeaver2/dataGeneration/trainmodel/train_lenet.py

The successful training generates the weights and storein. caffemodel format at
dnnWeaver2/dataGeneration/caffemodelzoo/lenet/
snapshot_20ch_50ch_500ch_10ch_lenet_iter_10000.caffemodel

5. Simulation Data Generation

The trained weight values for each layer of LeNet and the input image to be tested at DNNWeaver2 simulation environment needs to be pre-loaded to the DDR model in the testbench.
Below scripts are reading the image /. caffemodel files and generates the image and weights in the format required by DNNWeaver2 Systolic Array block and the DDR model.

Note: All data in Qmn format. Refer Appendix-1 for Qmn format and operations.

5.1 Generate input image data:

Use the following script to generate input image data,
dnnWeaver2/dataGeneration/genscripts/image_input_prep/run.sh

To generate different image data,select the test digit in run.sh
. /genImageData.py -i ../../dataset/mnist/test_data/7_img.bmp -o img_7.hex >& 7.log

This script will generate following files:

1) img_7_qmn_2d.dec : input pixel data of digit 7 is visualized in 2D decimal format.

2) img_7_qmn_2d.hex: input pixel data of digit 7 is visualized in 2D hex format.
3) img_7.hex: input pixel data used in DW2 simulations of digit 7 is visualized in 1D hex format.

Copy img_7.hex to simulation data folder
dnnWeaver2/dnnweaver2_lenet/simulation/data/inputimage.txt

5.2 Generate weight Data

To generate weights in the required format for DW2 use the following script,
/dnnWeaver2/dataGeneration/genscripts/weight_prep/run.sh

Command:
./genWtData.py-w ../../caffemodelzoo/lenet/snapshot_20ch_50ch_500ch_10ch_lenet_iter_10000.caffemodel -p ../../prototxt/lenet/lenet.prototxt -model lenet >& gen_weights.log

This script generates following files :
Layer Name: conv1 , Shape: (20, 1, 5, 5)
Generated weights in systolic array format in
data/lenet_20_50_500_10/conv1.float
data/lenet_20_50_500_10/conv1.hex

Layer Name: conv2 , Shape: (50, 20, 5, 5)
Generated weights in systolic array format in
data/lenet_20_50_500_10/conv2.float
data/lenet_20_50_500_10/conv2.hex

Layer Name: fc1 , Shape: (500, 1250)
Generated weights in systolic array format in
data/lenet_20_50_500_10/fc1.flt
data/lenet_20_50_500_10/fc1.hex

Layer Name: fc2 , Shape: (10, 500)
Generated weights in systolic array format in
data/lenet_20_50_500_10/fc2.flt
data/lenet_20_50_500_10/fc2.hex

*.hex files are for the use to simulate DNNWeaver2. *.flt files are generated to manually verify the format, if required.

copy **conv1.hex** to dnnweaver2_lenet/simulation/data/**whex_conv0.txt**
copy **conv2.hex** to dnnweaver2_lenet/simulation/data/**whex_conv1.txt**
copy **fc1.hex** to dnnweaver2_lenet/simulation/data/**whex_fc0.txt**
copy **fc2.hex** to dnnweaver2_lenet/simulation/data/**whex_fc1.txt**

6. DNNWeaver2 Compiler

6.1 DNNweaver2 LeNet graph

DNNweaver2 accepts the network model in tensorflow graph model. The Lenet graph is specified in the below file and input to the DNNWeaver2 compiler.
Path: dnnWeaver2/dnnweaver2_lenet/compiler/lenet_20_50_500_100_sys4x4.py

LeNet graph :

```
with graph.as_default():
    ## 28x28 grey image padded to 32x32
    with graph.name_scope('inputs'):
        i = get_tensor(shape=(batch_size,32,32,1), name='data', dtype=FQDtype.FXP16, trainable=False)    #32x32 ( padded
    ## conv0
    with graph.name_scope('conv0'):
        weights = get_tensor(shape=(20,5,5,1),                                          ## ic=1, oc =20
                             name='weights',
                             dtype=FixedPoint(16,7))                                    ## q9.7
        biases = get_tensor(shape=(20),
                             name='biases',
                             dtype=FixedPoint(32,20))
        conv = conv2D(i, weights, biases, pad='VALID', dtype=FixedPoint(16,8))
        pool = maxPool(conv, pooling_kernel=(1,2,2,1), stride=(1,2,2,1), pad='VALID')
    ## conv1
    with graph.name_scope('conv1'):
        weights1 = get_tensor(shape=(50,5,5,20),                                                ##  ic=20, oc =50
                             name='weights1',
                             dtype=FixedPoint(16,7))
        biases1 = get_tensor(shape=(50),
                             name='biases1',
                             dtype=FixedPoint(32,20))
        conv1 = conv2D(pool, weights1, biases1, pad='VALID', dtype=FixedPoint(16,8))
        pool = maxPool(conv1, pooling_kernel=(1,2,2,1), stride=(1,2,2,1), pad='VALID')
   ## fc0
    with graph.name_scope('fc0'):
        cout = pool.shape[-3]
        hout = pool.shape[-2]
        wout = math.ceil(pool.shape[-1]/num_rows)*num_rows
        size = cout*hout*wout
        in_fc= get_tensor(shape=(1,1,1,size),name='fcin',dtype=FixedPoint(16,8),trainable=False)
        weights1 = get_tensor(shape=(500,1,1, size),                                    ## neurons =500
                             name='weights1',
                             dtype=FixedPoint(16,7))
        biases1 = get_tensor(shape=(500),
                             name='biases1',
                             dtype=FixedPoint(32,20))
        fc0 = conv2D(in_fc, weights1, biases1, pad='VALID', dtype=FixedPoint(16,8))
    ## fc1
    with graph.name_scope('fc1'):
        cout = fc0.shape[-3]
        hout = fc0.shape[-2]
        wout = math.ceil(fc0.shape[-1]/num_rows)*num_rows
```

6.2 Generate DNNWeaver2 instruction set

cd dnnWeaver2/dnnweaver2_lenet

Step1: Set Tensor Flow paths & env variables
$> source evn/.rctensorflow
cdnserv2> source env/.rctensorflow
(tfpy36cpuenv) cdnserv2>

Step2 : Generate Instructions
1) cd compiler
2) lenet model is specified in "lenet_20_50_500_100_sys4x4.py" file
3) run compiler -> ./run.sh

This Generates the instructions for the specified network.

4) Outputs:
- instruction.bin (in hex format)
- compiler.log (contains logs and ddr address for each layer)

6.3 Simulate LeNet in DNNWeaver2

1) copy instruction.bin & compiler.log from "compiler" folder to simulation folder
2) cd simulation
3) In rtl/rtl/dnnweaver2_controller.v ensure that (ARRAY_M,ARRAY_N)= (num_rows,num_cols) as in graph.

lenet_20_50_500_100_sys4x4.py:
num_rows = 4
num_cols = 4

.v:
```
module dnnweaver2_controller #(
    parameter integer  NUM_TAGS                 = 2,
    parameter integer  ADDR_WIDTH               = 42,
    parameter integer  ARRAY_N                  = 4,
    parameter integer  ARRAY_M                  = 4,
```

4) Open tb/axi_master_tb_driver.v
- Open compiler.log & search for string "Addr --"
- For each Layer there will be entry for Addr (Data,Weight,Bias & output). For eg.

```
INFO:Graph Compiler:MS : In Layer :fc1/Convolution
INFO:Graph Compiler:MS : Addr -- Data : 0xf000       Weights : 0x10000
INFO:Graph Compiler:MS : Addr -- Bias : 0x11000      Outputs : 0x12000
```

- line 202 in "tb/axi_master_tb_driver.v" file update these address locations for all layers into the paramaters. (from run.log/compiler.log file)

5) Now execute simulation ./sim.sh

This runs the iverilog simulaion with the below entries.

```
> iverilog  -c filelist.tb -g2012  -o compiler.out -Dsimulation -I .  -s tb_dnnw2_ctrl
```

6) Output :
   - sim.log
   - tb_dnnw2_ctrl.vcd (vcd sump)

7) Open gtkwave. The fc_layer.gtkw contains the basic signal of interest.

```
$>gtkwave tb_dnnw2_ctrl.vcd fc_layer.gtkw &
```

8) Validation

Here, input image digit is 7 which havehighest value neuron in FC2 layer output. The classified output details are available in sim.log

FC2 Final Output	decimal_format	hex_format
ddr_ram[0]=	177	00000000000000b1
ddr_ram[1]=	64936	000000000000fda8
ddr_ram[2]=	130	0000000000000082
ddr_ram[3]=	579	0000000000000243
ddr_ram[4]=	64033	000000000000fa21
ddr_ram[5]=	329	0000000000000149
ddr_ram[6]=	64180	000000000000fab4
ddr_ram[7]=	1995	00000000000007cb(Classified Digit)
ddr_ram[8]=	65292	000000000000ff0c
ddr_ram[9]=	647	0000000000000287
ddr_ram[10]=	0	0000000000000000
ddr_ram[11]=	0	0000000000000000

7. DDR Image /Weight size Calculations for Simulation

The testbench Loads the initial input image and weight data to the input DDR memory. To perform this operation few parameters needs to be modified in the testbench based on the network and layer values outputs.

Layer Name / Parameter	Size	Remarks
Systolic Array Size	4x4	
ARRAY_N	4	4 IC parallel channels
ARRAY_M	4	4 OC parallel Channels
Input Image	28x28	Input Image 32x32(after padding)
Kernel and Stride	5x5, 1	
Conv1	1 (ic), 20 (oc),	Output Image Size = 28x28.(32-5+1)
MaxPool1	2x2 ,2	Output Image Size =14x14 (28/2)
Conv2	20 (ic), 50 (oc)	Input Image size = 14x14 Output Image size = 10x10 (14-5+1)
MaxPool2	2x2 ,2	Output Image = 5x5 (10/2)
FC1	50 (ic), 500 (oc)	500 Neurons
FC2	500 (ic) , 10 (oc)	10 Neurons

The systolic array is 4x4, so 4 input channels can be applied simultaneously.So, the input image size to be loaded to DDR is “input image size” x ARRAY_N.

Input image size: 4096 locations

Padded digit input image size => 32 x 32.
DDR Image size = Input Image Size * ARRAY_N, => 32*32*4 = 4096

Conv1 Weight size: 2000 locations.
To compute the total DDR weight size, we need to consider the systolic array size, input feature maps, output feature map and kernel size.
For Conv1, the number of input channel is 1 and output channels is 20.
The Weight size for one cycle of systolic array size 4x4 (Kx * ky * ARRAY_N * ARRAY_M).
= (5 * 5 * 4 * 4) = 400.

To compute 20 Output Feature maps, the no of cycles or iterations in systolic array is Ceil (20 / ARRAY_M) = 5.
To use 1 input feature map the no of cycles or iterations in systolic array is ceil (IC/ARRAY_N) = ceil (1 / 4) = 1 ;
So, the total weight size is (1*5*400) = 2000 locations

Maxpool Image Size is 14x14 and stored in DDR. This is padded and feed as input for the next layer.

Conv2 input image size:

Padded image, input image + kernel size – 1: 14+5-1=18.
DDR image size 18*18*4 = 1296, where 4 is ARRAY_N size.

Conv2 weight size:
Each systolic array takes 400 (5*5*4*4) weight size, whereas 5*5 is kernel size and 4*4 is systolic ARRAY_M x ARRAY_N.

20 Input channels = ceil (20 / ARRAY_N) = ceil (20/ 4) = 5
50 output channels = ceil (50/ARRAY_M) = ceil (50/4) =13
To compute 20 IC's and 50 OC's, it takes (5*13*400) = 26000 weight size

After Maxpool2, image size will become 5x5 which is input to FC1 layer

FC layers:
2D Output of Conv+Pool is converted to single 1D array. This array is the input to all the neuron in the FC layer and another input is the weight matrix. Each element of the input 1D array can be considered as one input channel for the systolic array. So, each element is now a 1x1 image. The corresponding weight kernel size is also 1x1. For each node, we have N input channels and N weight kernels of size 1x1.

FC1:

```
with graph.name_scope('fc0'):
    cout = pool.shape[-3]
    hout = pool.shape[-2]
    wout = math.ceil(pool.shape[-1]/num_rows)*num_rows
    size = cout*hout*wout
    in_fc= get_tensor(shape=(1,1,1,size),name='fcin',dtype=FixedPoint(16,8),trainable=False)
    weights1 = get_tensor(shape=(500,1,1, size),                    ## neurons =
```

No. of IC = 50 (Output of conv2, before flattening)
No. of OC = 500;
Systolic array size = 4x4;
Input image: 5x5 (Before flattening)
Kernel: 5x5 (before flattening)
Stride: 1
IMGSIZE = 1300 (Input to FC1, after flattening)
WGHTSIZE = 650000

FC1 Image size (IMGSIZE)
To compute 50 IC's, we need 13 (50/4 + 1) sets of systolic array. Each IC has 25 elements.
So image size is 13*25*4 =1300

```
27      fc0/fcin[1,1,1,1300] (FXP16 (8,8))
28      fc0/weights1[500,1,1,1300] (FXP16 (9,7))
```

FC1 weight size (WGHTSIZE)
Each set of systolic array has 4 IC and 4 OC. To compute 50 IC's, we need 13 sets of systolic arrays. To compute 500 OC's, we need 125 sets of systolic arrays.
Each systolic array takes 400 (5*5*4*4) weight sizes. Here 5*5 is kernel size and 4*4 are systolic ARRAY_M x ARRAY_N.

To compute 50 IC's and 500 OC's, it takes (13*125*400) = 650000 weight size

Compiler:
NumOutputChannels (after padding) = ceil(NumOutputChannels/M)*M = ceil(1/4)*4 = 4
size = cout*hout*wout
in_fc = get_tensor(shape=(1,1,1,size),name='fcin',dtype=FixedPoint(16,8),trainable=False)

2.FC2:

```
## fc1
with graph.name_scope('fc1'):
    cout = fc0.shape[-3]
    hout = fc0.shape[-2]
    wout = math.ceil(fc0.shape[-1]/num_rows)*num_rows
    size = cout*hout*wout
    in_fc= get_tensor(shape=(1,1,1,size),name='fcin',dtype=FixedPoint(16,8),trainable=False)
```

No. of IC = 500;
No. of OC = 10;
Systolic array size = 4x4;
Kernel: 1x1
IMGSIZE = 500
WGHTSIZE = 6000

To compute image size(IMGSIZE):
Each element of input 1D array is considered as 1 input channel. Here it has 500 IC's ,so image size is 500.

```
32      fc1/fcin[1,1,1,500] (FXP16 (8,8))
33      fc1/weights1[10,1,1,500] (FXP16 (9,7))
```

To compute weight size(WGHTSIZE):
Each set of systolic array has 4 IC and 4 OC. To compute 500 IC’s, we need 125 sets of systolic arrays. To compute 10 OC’s, we need 3 sets of systolic array.
Each systolic array takes 16 (1*1*4*4) weight sizes. Here 1*1 is kernel size and 4*4 is systolic ARRAY_M x ARRAY_N.

To compute 500 IC's and 10 OC’s, it takes (125*3*16) = 6000 weight size

```
/// In Layer :conv0/Convolution
parameter IMG    = 32'h0000>>1;
parameter WEIGHT= 32'h8000>>1;
parameter BIAS   = 32'hc000>>1;
parameter OP     = 32'hd000>>1;

parameter IMGSIZE=4096;
parameter WGHTSIZE=2000;

//// In Layer :conv1/Convolution
parameter IMG2    = 32'ha7000>>1;
parameter WEIGHT2= 32'haf000>>1;
parameter BIAS2   = 32'he2000>>1;
parameter OP2     = 32'he3000>>1;

parameter IMGSIZE2=1296;
parameter WGHTSIZE2=26000;


// In Layer :fc0/Convolution
parameter IMG3    = 32'h116000>>1;
parameter WEIGHT3= 32'h119000>>1;
parameter BIAS3   = 32'h60f000>>1;
parameter OP3     = 32'h611000>>1;

parameter IMGSIZE3   = 196;
parameter WGHTSIZE3 = 650000;

// In Layer :fc1/Convolution
parameter IMG4    = 32'h615000>>1;
parameter WEIGHT4= 32'h616000>>1;
parameter BIAS4   = 32'h622000>>1;
parameter OP4     = 32'h623000>>1;

parameter IMGSIZE4=500;
parameter WGHTSIZE4=6000;
```

Figure 3: axi_master_tb_driver.v file entry

Specify the computed Image and weights and each layer address in /dnnWeaver2/dnnweaver2_lenet/simulation/tb/axi_master_tb_driver.v

8. Expected Output Calculations (in Qmn data format)

Use the below script to generate the expected convolution/maxpool/FC layer outputs in qmn format.

Path:/svn/FPGACNNProject/dnnWeaver2/dataGeneration/genscripts/lenet_layerresults_compute

Command: ./run.sh

This script generates the *.log file, includes layer wise computations which is used to verify the DDR output.

To compute results for different images and different lenet prototxt configurations, load the *.bmp image in the run.sh.

./genResults.py -i ../../dataset/mnist/test_data/7_img.bmp -ptxt ../../prototxt/lenet/lenet.prototxt-w ../../caffemodelzoo/lenet/snapshot_20ch_50ch_500ch_10ch_lenet_iter_10000.caffemodel>& img7_layerresult_qmndec.log

9. Simulation Data Validation

The simulated data in DNNWeaver2 is validated against the layer wise computed data as generated in section 8. However, the entire computation in the simulation and expected data calculation is in QMN format.

So, the QMN format data needs to be converted back to floating value and compared.

Software Data:

The below pictures show the Qmn value to floating value conversion required for the software layers data.

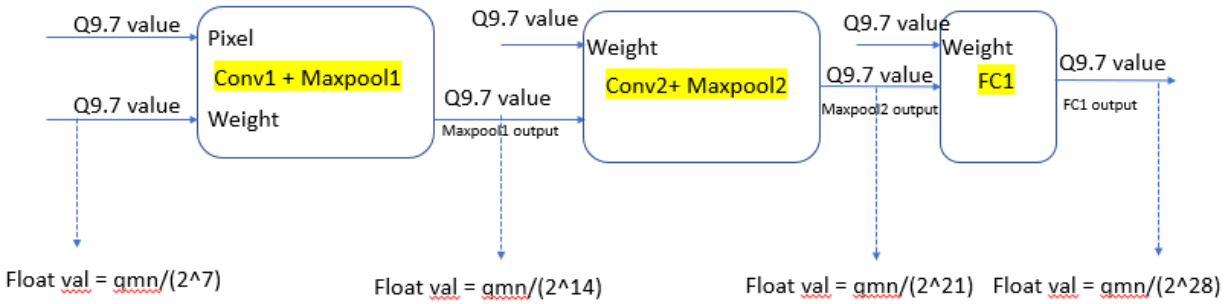

Figure 4 : Q9.7 to floating conversion Process for Software Data

For example, in “img7_layerresult_qmndec.log”, the expected FC2 output is specified in qmn format as below

[47421684473 -161626568891 36044273723 155717563951 -403610143488 88412703520 -364351692637 535969735536 -64476793451 172600868425]

To get the equivalent floating value divide the corresponding output (qmn) number by 2^35. i.e 535969735536 / 2^35 = 15.5987

Simulation Data:

In compiler graph we can specify the Qmn value in “dtype=FixedPoint(16,7))” . The results stored in ddr memory is 16-bit data sliced [22:7] from a 64-bit accumulated data.

So, output of each layer stored in DDR is always divide by 2^7.

In sim.log, FC2 output of 7th neuron (digit 7) is ‘1995’ in Qmn format. The floating value is 1995/ 2^7 =15.5859.

The data value in simulation 15.5859 is close to the software computed expected data 15.5987. The accuracy loss in comparison is accepted.

In this way the simulation data is manually compared with expected data across the layers to validate the functional correctness of dnnweaver2.

Appendix -1

QMN format

Qmn is a binary fixed-point format where the number of fractional bits and the number of integer bits is specified. Q format is often used in hardware that does not have a floating-point multiplier unit.

Example Qmn formats:

Q16 is the format where 16 fractional bits are represented in 16bit data size
Q1.15 represents 1 integer and 15 fractional bits in the 16bit data size.
Q8.8 represents 8 integer and 8 fractional bits in the 16bit data size.
Q9.7 represents 9 integer and 7 fractional bits in the 16bit data size.

Convert a fractional number to Q9.7 format

Decimal fraction: 0.44722712
⇨ 0.44722712 * 2^7 (where 7 is the fractional bit portion)
⇨ 57.2450
⇨ Ignore fractions
⇨ (57)d
⇨ Convert 57 to Hex
⇨ *(0039)h*

Below are few sample conversion

Num: 0.44722712: Q9.7: (57)d , 0039h
Num: 0.38406804: Q9.7: (49)d , 0031h
Num: 0.21279903: Q9.7: (27)d , 001Bh
Num: 0.005641659: Q9.7: (0)d , 0000h
Num: -0.20501006: Q9.7: (-26)d , Do 2's complement TC: 1111111111100110 , FFE6h
Num: 0.01629672: Q9.7: (2)d , 0002h
Num: 0.15990347: Q9.7: 20 , 0014

Multiplication of 2 Qmn numbers (pos x pos):

Num: 0.15990347: Q9.7: (20)d , 0014h
Num: 0.23478635: Q9.7: (30)d , 001Eh
⇨ 0014h x 001Eh
(0258)h

Convert Qmn to Fraction
⇨ (0258)h / 2^14 (14 = 2 times the fractional bit n)
(600)d / 2^14
⇨ 0.03662109375
Verification:
(0.15990347) d * (0.23478635) d = 0.03754315207
– Minor Accuracy Loss
RTL approach :
In rtl design, for multiplication of two no's (16 bitsx16 bits =32 bits) output is assigned to 16

bits, considering the portion of out bits starting from 7th bit till 22nd bits of 0285h.

0014h x 001Eh = 0004h ([22:7] bits of 0258)

0258 = 0000 0000 0|000 0000 0000 0010 0|101 1000

out = 0000 0000 0000 0100 = (0004)h = (4)d

➔ (4)d / 2^7 (as output is 16 bit, so divide by 2^7)

➔ 0.03125

Multiplication of 2 Qmn numbers (neg x pos):

Num: -0.3230421 : Q9.7: (-41)d , Do 2's complement TC: 1111111111010111 , FFD7

Num: 0.25686562 : Q9.7: (32)d , 0020h

- ⇨ (FFD7) h * (0020) h
- ⇨ 1F FAE0
- ⇨ Overflow the 16 bits ;
- ⇨ Ignore the overflow data and consider FAE0 only
- ⇨ Overflow indicates negative number
- ⇨ Do two's complement of FAE0
- ⇨ (010100100000)2
- ⇨ (1312)d
- ⇨ Convert Qmn results to Fraction
- ⇨ (1312)d / 2^14
- ⇨ 0.080078125
- ⇨

Verification :

-0.3230421 * 0.25686562 =>-0.0829784093

rtl approach :

For neg*pos no's , take 2's compliment of output which is again 16 bit and divide the resultant decimal number by 2^7.

FFD7h x 0020h = fff5

fff5 = 1111 1111 1111 0101 (MSB bit is 1 , ouput is -ve)

2's comp = 0000 0000 0000 1011 = (-11)d

➔ (11)d / 2^7

➔ 0.0859375

Appendix – 2

Required Tool Chains

1. Caffe framework tools (for training and data gen scripts)
2. Python 3.6
3. iverilog & gtkwave (for simulation and waveform viewing)