ECSE 682 VLSI for Machine Learning

Project Report

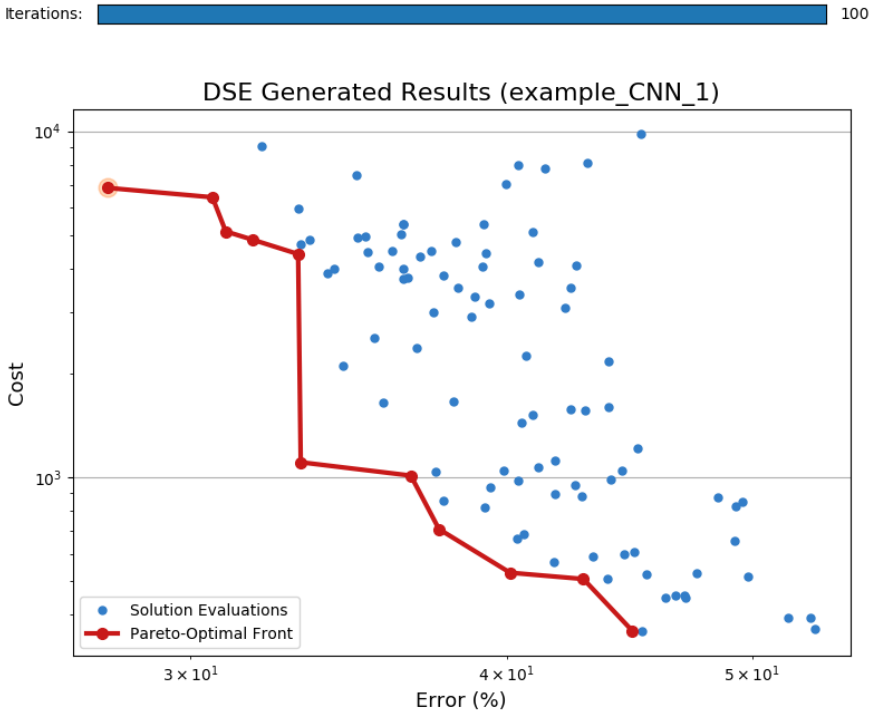
Zonghao Li # 260787179

Part I: CNN network training using OPAL

Software `OPAL` is used to train the CNN network, and the following CNN network architecture is selected whose misclassification rate is about 28% (better candidates should be selected with longer training time, in this case it does not affect the implementation however):

```
#        Shape of element 0: (5, 5, 3, 32)
#        Shape of element 1: (32,)
#        Shape of element 2: (32,)
#        Shape of element 3: (32,)
#        Shape of element 4: (32,)
#        Shape of element 5: (32,)
#        Shape of element 6: (3, 3, 32, 64)
#        Shape of element 7: (64,)
#        Shape of element 8: (64,)
#        Shape of element 9: (64,)
#        Shape of element 10: (64,)
#        Shape of element 11: (64,)
#        Shape of element 12: (7, 7, 64, 64)
#        Shape of element 13: (64,)
#        Shape of element 14: (64,)
#        Shape of element 15: (64,)
#        Shape of element 16: (64,)
#        Shape of element 17: (64,)
#        Shape of element 18: (256, 10)
#        Shape of element 19: (10,)
#        Shape of element 20: (10,)
#        Shape of element 21: (10,)
#        Shape of element 22: (10,)
#        Shape of element 23: (10,)
```

In this network, there are 3 CNN layers followed by a fully connected layer. The first CNN network has 32 kernels (filters) and in total 3 channels, corresponding to RGB colors. Each kernel is 5-by-5 in dimension. The input samples are `CIFAR10`, which are 10000 32-by-32 RGB pictures. The second element (`element 1`) is the bias vector for the first layer. `element 1` to `element 5` are batch normalization vectors. In this project they are not considered. The second layer has 64 filters with 32 channels, and each filter is 3-by-3 in size, and `element 7` is the bias vector for the second layer. The same parsing approach can be taken to the third layer and the fully connected layer. These can be visualized by OPAL, even though the figure is not very intuitive.



Selected: 20
Error: 0.2783
Cost: 6853
fc_layers: [], conv_filters: [32, 64, 64], conv_pools: [2, 2, 1], conv_kernels: [5, 3, 7], conv_strides: [1, 2, 3], learning_rate: 0.006634320132434368

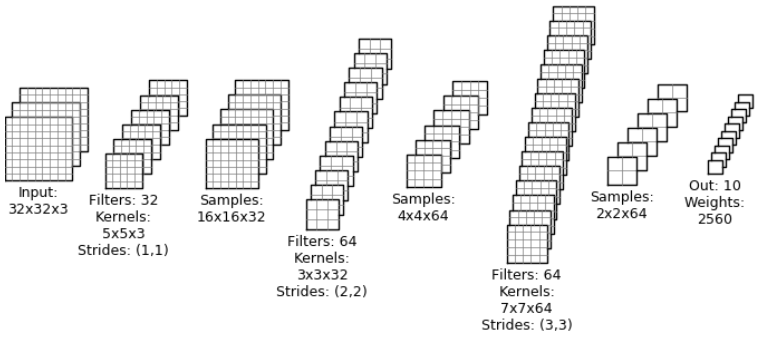


Figure 1 Selected CNN network

Part II: Results parsing and fixed-point CNN computations

The data structure of the results generated by OPAL is not really following the convention. For instance, as noted in [1] and [2], the dimension of a kernel (weight) tensor should be denoted as the following:

$$\boldsymbol{W}[u][k][i][j]$$

Where u is the number of kernels, k is the dimension of the channel, i is the row dimension for each 2D kernel plane, and j is the column dimension for each 2D kernel plane. This can be visualized by the following figure:

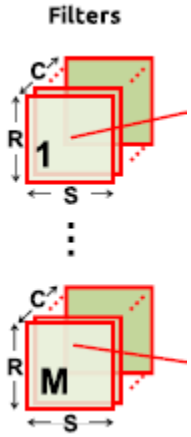


Figure 2 Tensor dimension notation convention [1]

Therefore, the reshaping of the obtained results is desired, both in the software and hardware implementation, to make design relatively more straightforward, even though it might not be necessary. For CNN computation of a layer, the following equation is employed [1]:

$$\boldsymbol{O}[z][u][y][x] = ReLU(\boldsymbol{B}[u] + \sum_{k=0}^{C-1}\sum_{i=0}^{R-1}\sum_{j=0}^{S-1} \boldsymbol{I}[z][k][Ux+i][Uy+j] \times \boldsymbol{W}[u][k][i][j])$$

$$0 \le z < N, 0 \le u < M, 0 \le y < E, 0 \le x < F, E = \frac{H-R+U}{U}, F = \frac{W-S+U}{U}$$

Where $\boldsymbol{O}$, $\boldsymbol{I}$, $\boldsymbol{W}$ and $\boldsymbol{B}$ are output feature maps (ofmap), input feature maps (ifmap), weights (kernels or filters), and biases. U is the stride number (1 for first layer, 2 for second layer, and 3 for third layer in the generated CNN network given by $Fig.1$), R is the row dimension for each 2D kernel, S is the column dimension for each 2D kernel, H is the row dimension for each 2D input feature map, W is the column dimension for each 2D input feature map, C is the channel dimension for input filters and input feature maps, M is the numbers of kernels and the channel

dimension for the output feature maps, E is the row dimension of the output feature maps, and F is the column dimension of the output feature maps . This procedure can be visualized in $Fig.\,3$.

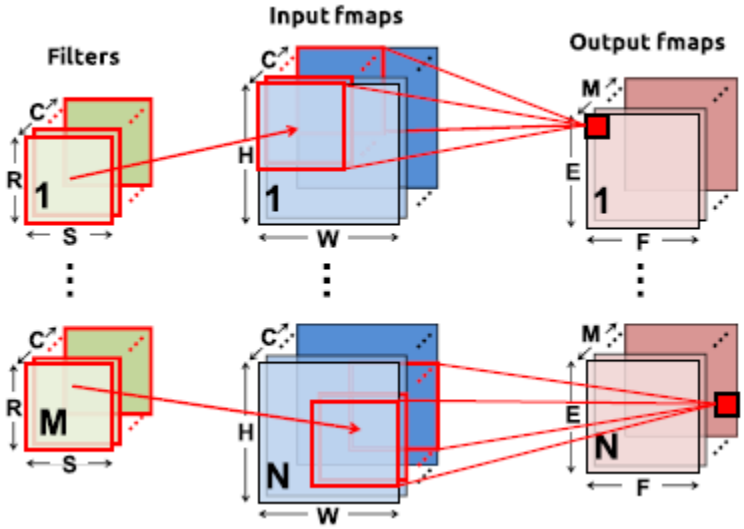


Figure 3 CNN computation in a layer [1]

With the aid of this, we can generate the dimension table for all the layers given by OPAL (in this project, only CNN part will be considered, and no zero-padding, pooling, batch normalization is into account, so only bias matrices are presented), which are listed in $Table\ 1$.

Table 1 Summary for all CNN parameters

	Filter 1	Input 1	Output 1	Filter 2	Input 2	Output 2	Filter 3	Input 3	Output 3
N (input and output numbers)	-	10000	10000	64	10000	10000		10000	10000
M (filter numbers and output feature map channel)	32	-	32	64	-	64	64	-	64
C (channel dimension for input and kernel)	3	3	-	32	32	-	64	64	-
R (row dimension for kernel)	5	-	-	3	-	-	7	-	-

S (column dimension for kernel)	5	-	-	3	-	-	7	-	-
H (row dimension for input)	-	32	-	-	28	-	-	13	
W (column dimension for input)	-	32	-	-	28	-	-	13	
E (row dimension for output)	-	-	28	-	-	13	-	-	3
F (column dimension for output)	-	-	28	-	-	13	-	-	3
U (Stride)	1			2			3		

A fixed-point CNN model is constructed in `Python`. And correspondingly a `Matlab` script is also written for the conversion from fixed-point decimals to fixed-point binaries in the vector format so that they are hardware compatible. In this project, 16-bit word length is selected as it is a common choice for CNN implementations that ensures the accuracy and efficiency simultaneously.

The `Python` script in $Appendix\ I$ does the tensor reshaping first, and then the fixed-point ($Q8.8$) CNN computations based on the equation provided in [1] are conducted over three layers. The fully connected layer here is neglected here as it is not a part of this project, but it could have been easily adapted to the already existed architecture. The `Matlab` script can be found in $Appendix\ II$.

Convolutional layer calculations require lots of computational powers, the attached `Python` script is not computational efficient.

Part III: Hardware implementation of CNN accelerator

Alternatively, a processing element (PE) array architecture can be established in hardware level. Guided by [1], the first CNN convolutional layer structure can be found in $Fig.\,4$.

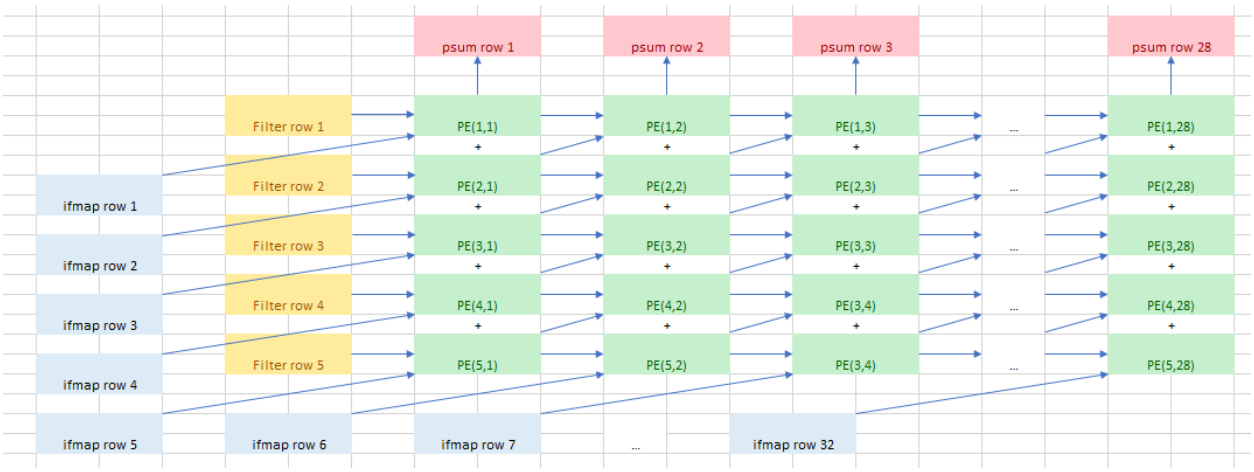


Figure 4 A 5-by-28 "row stationary" PE array for one channel CNN computation, blue arrows here mean data and transported within the same clock cycle

The dimension of this PE array is determined by the kernel row dimension R, input feature map row dimension H, and the output feature map row dimension E, which are 5, 32, and 28. Therefore, this architecture has 5 by 28 PEs. Inside each PE, for illustration purposes, say $PE(1,1)$, the row stationary method for the kernel will be used, therefore only the input feature maps are updated in each clock cycle, determined by stride U:

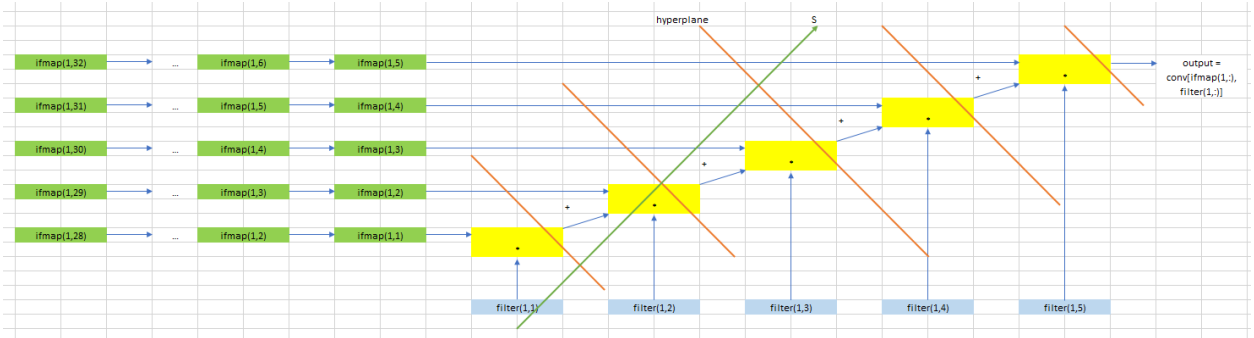


Figure 5 Systolic architecture inside the first PE for one channel CNN computation

Systolic structure will be applied to each one of the PE ($Fig.\,5$). Here, the orange line is denoted as the hyperplanes, and the green arrow represents schedule vector $\boldsymbol{S}$. In every 5 clock cycle, for $PE(1,1)$, the output will be the partial-sum of the first row of the first filter (in the first channel) convolves with the first five elements from the first row of the first input feature map (in the first

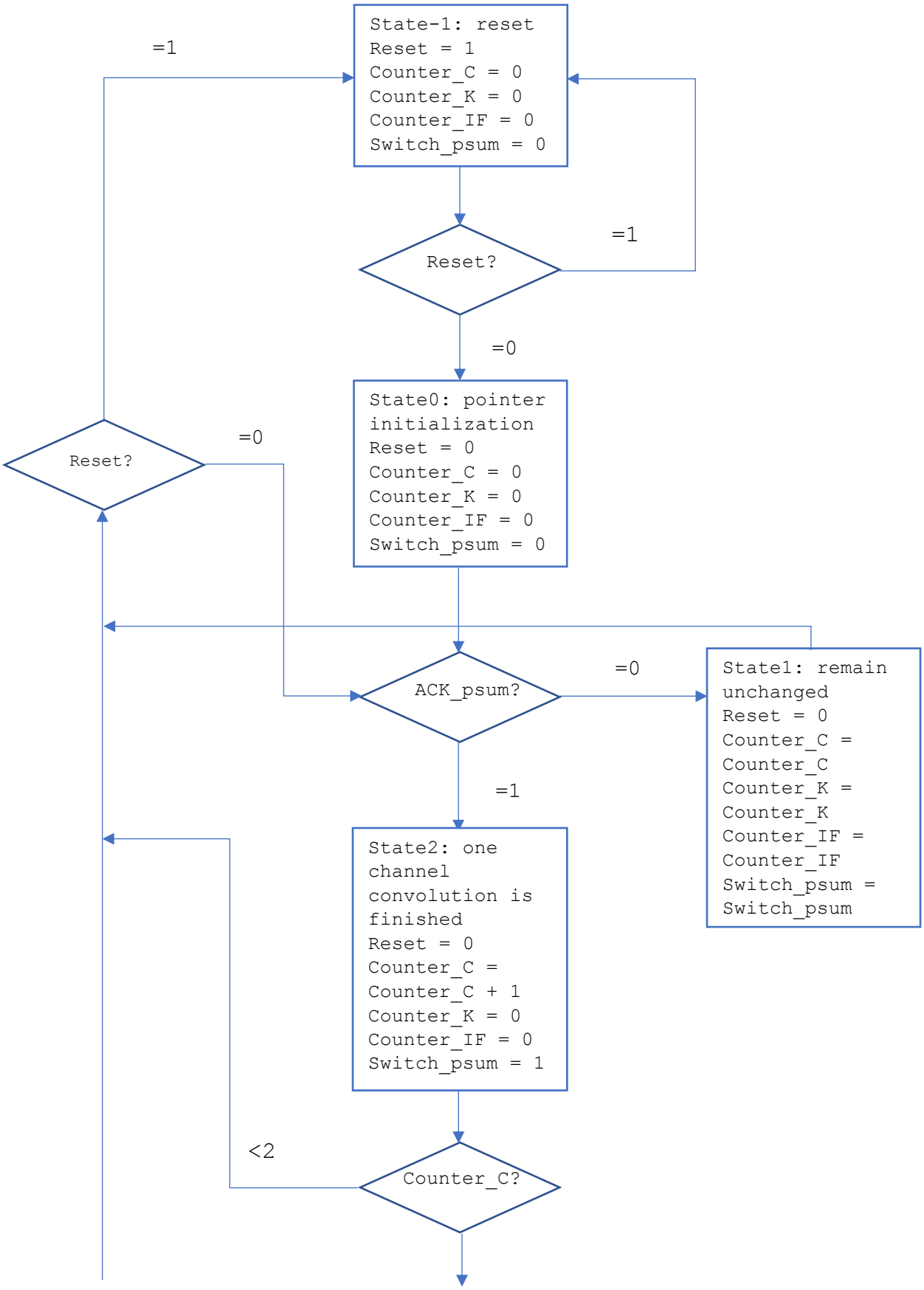
channel). After that, the kernel will be fixed but the first row of the first input feature map will be sifted by $U = 1$, and the same convolution will be conducted. By this manner, to collect 28 partial-sum outputs from the convolution of the first row of the first filter to the first row of the first input feature maps, there will be in total $5 \times 28 = 140$ clock cycles required, representing the hyperplane in $Fig. 4$. Therefore, for the generation of the first row of the output feature maps, at least there will be in total $140 \times 5 = 700$ clock cycles, without counting the summation of the partial-sum generated by the convolution results from each row of one kernel. Since there are 3 channels for the kernel and input feature maps in the first layer, for generating the first output feature map in the first channel, due to the parallel structure of the systolic array, in total at least $700 \times 3 = 2100$ clock cycles are required; for generating one entire output feature map, since there are 64 filters, in total $64 \times 2100 = 13440$ clock cycles required. Entirely there are 10000 pictures, so $13440 \times 10000 = 1.34G$ clock cycles. For multi-channel CNN computation, this computation procedure can be repeated. The same methodology can be applied to the second and third layer.

The `Verilog` implementation of single layer CNN convolution can be referred to $Appendix\ III$. Note that as the input output flow for `Verilog` does not support MIMO, so parallel I/O must be converted to serial manner, which is very insufficient. However, the already designed CNN computation layer can be easily adapted to multilayer structure in `SystemVerilog` since it supports parallel I/O flow mechanisms. Therefore, to be clear about the working flow of one single CNN convolutional layer is more crucial.

The Algorithm State Machine (ASM) diagram below guides the architecture construction of the single `Verilog` CNN convolutional layer; particularly, it is for the first layer of the selected network, but again, this methodology is applicable and easily adaptable to any other multiple layer structure. In $Fig. 6$, there are some important state indicators: `Reset`, `Counter_C`, `Counter_K`, `Counter_IF`, and `ACK_psum`. Their governing functionalities can be summarized in $Table\ 2$.

Table 2 Some important state indicators

`Reset`	Resetting the program (including resetting all the pointers back to "0")
`Counter_C`	Counting the channel, if it is "2", it means one kernel is finished, then it is reset back to "0"
`Counter_K`	Counting the finished kernel; triggering once `Counter_C` reset from "2" to "0"
`Counter_IF`	Ifmap counter, triggers once both `Counter_C` and `Counter_K` are reset
`ACK_psum`	Acknowledgement signal


State-1: reset
Reset = 1
Counter_C = 0
Counter_K = 0
Counter_IF = 0
Switch_psum = 0
=1
Reset?
=1
=0
State0: pointer initialization
Reset = 0
Counter_C = 0
Counter_K = 0
Counter_IF = 0
Switch_psum = 0
Reset?
=0
ACK_psum?
=0
State1: remain unchanged
Reset = 0
Counter_C = Counter_C
Counter_K = Counter_K
Counter_IF = Counter_IF
Switch_psum = Switch_psum
=1
State2: one channel convolution is finished
Reset = 0
Counter_C = Counter_C + 1
Counter_K = 0
Counter_IF = 0
Switch_psum = 1
<2
Counter_C?

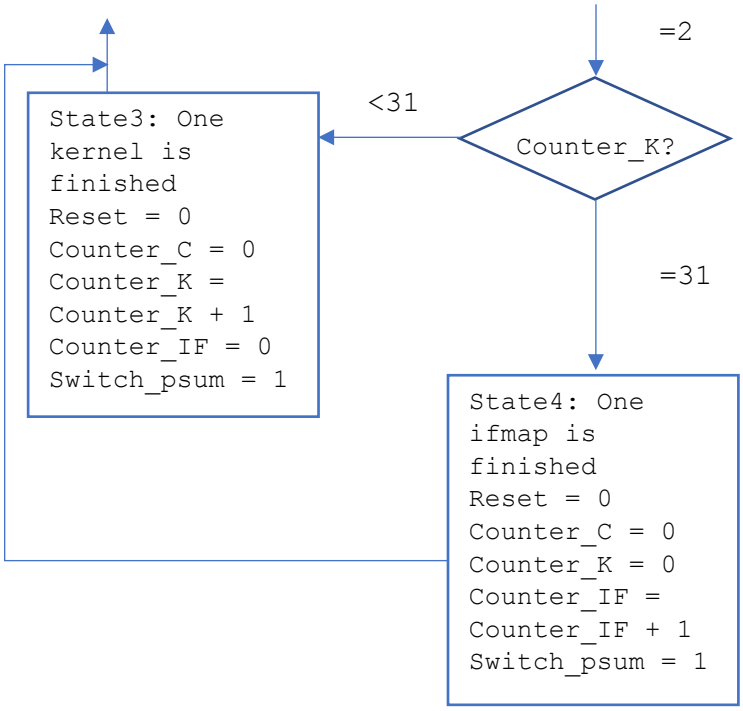


Figure 6 ASM diagram of a CNN convolutional layer

Special explanation here will be given to the acknowledgement signal `ACK_psum` since it is the handshaking triggers among all PEs and it controls the working flow of this architecture. `ACK_psum` will only trigger once all PEs are done with the current convolutions of the kernel row and ifmap row, defined by `State0`. However, no matter when, if there is a switching happening on `ACK_psum`, it means there is one channel of the convolution between one row and one ifmap is finished; and since all PEs are conducted simultaneously, the acknowledgement signal is therefore synchronous: as long as there is one `ACK_psum` triggers, all PEs complete the current convolutions jobs. Then the partial summation will be conducted, as `Switch_psum` is now "1", triggered by `ACK_psum`. And the accumulation will continue until it reaches `Counter_C = 2` and `ACK_psum = 1` where one ofmap channel of the respective ifmap is completed. Then the bias adding and `ReLU` function will conduct to the accumulation results, and kernel pointers will be reset back to 0.

Accumulation is done by column scanning: as each time only one output can be generated from each PE, there is an addressing signal `Addr_psum` controlling the interested accessing data. It can be defined from 0 to 27 since each PE supposed to generate an arrow of one ofmap, as indicated by $Fig. 7$, which is the outcome of the "row stationary" method. Since each time `Addr_psum` increment by 1, one column of the ofmap of is finished (but if ifmap has multiple channel, then from one ofmap channel it needs to accumulate with the results from other

convolutions done by the other channels), therefore it requires at least 28 clock cycles to complete the one partial summation calculation. Also, as all calculations are conducted simultaneously, the same address location is defined to all PEs, and the data coming from the internal register defined by `Addr_psum` are retrieved for the calculation. This can be also found in *Fig*. 7, where `OUT_PE` is the output collection from all PEs (there are 140 PEs in this case), and `Addr_psum` keeps increasing until it reaches 27, where it is reset and then `Switch_psum` is also reset back to 0.

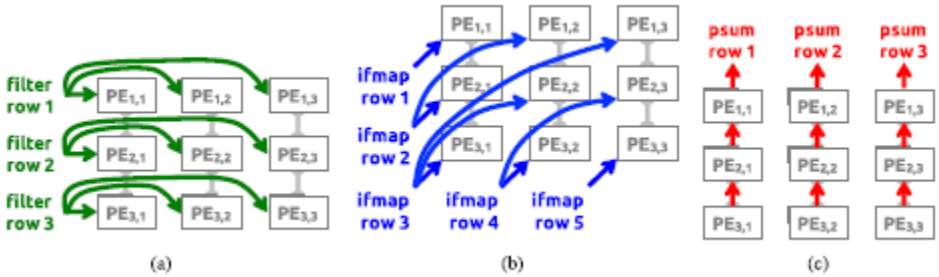


Figure 6 An example of the working flow of a PE array [1]

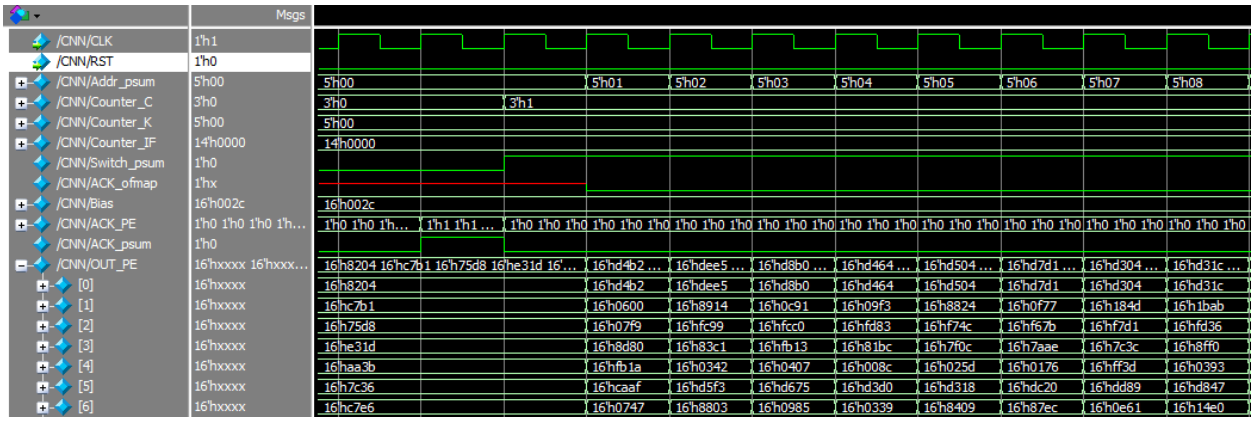


Figure 7 A snapshot of the PE data accessing mechanism of the designed CNN convolutional accelerator

Part IV: Synthesis Results and Comparisons

For each single CNN convolutional layer designed in this project, the throughput and latency is dominated by the size of the PE array. If the I/O data is inputted and outputted serially, then pipelining is required in order to avoid data corruption, which is really not efficient. Therefore, at input side, parallel data accessing is realized by using PE array, and output side the resulted data should also be accessed in parallel, this cannot be done in `Verilog` as it is not supported, so moving to `SystemVerilog` will be a good solution to support parallel data outputting.

In this design, assume the module is synthesised on Xilinx Virtex-7 xc7vx485tffg1157-1. Say the clock speed is running at 10MHz (default in Modelsim), each ofmap will take $(5 + 32) * 3 * 32 = 3552$ clock cycles, where the first "32" is the pipelining clock cycles for the delaying of PE array so that partial summation process will not be corrupted and the second "32" is the amount of filters. If the output result is transmitted to the next layer through serial communication, then an extra $(28 * 28 * 32) = 25088$ clock cycles are required (where "28" is the row and column dimension of the ofmap, determined by $E = \frac{H-R+U}{U}$), hugely slowing down the computation efficiency. In this case, a throughput 399 ofmap/sec throughput is calculated based on 10MHz clocking speed, and for 10000 `CIFAR10` this layer need to take 25.06 second to finish. Also, the latency is calculated by 0.0025 sec/ofmap.

The the utilization report of the synthesis can be found at $Appendix\ IV$ and for single layer with different dimension, the resources utilization will be altered majorly by the size of PE array, which is dominated by the kernel size and ofmap size.

For instance, for different layer size of the networks, with the same stride $U = 1$, same channel size $C = 3$, and same ifmap dimension $H = W = 32$, their corresponding utilization on the same FPGA chip can be found in $Table\ 3$.

Table 3 Utilization report for different PE array size

PE array	FF	LUT	Kernel amount	Throughput (ofmaps/sec) (10MHz clock speed and serial communication)
5×28 (current optimal selection)	1960	2660	32	399
3×30 (selection #8 from $Fig.\ 1$)	644	874	16	625
7×26 (selection #2 from $Fig.\ 1$)	2230	2955	64	201

Part V: Conclusion

It can be concluded that a single CNN convolutional layer is designed in hardware which works well in system level. Due to the limitation of time, the multi-CNN-layer structure is not realize in this design. However, this work can be easily adapted in the multi-layer structure, either in serial or parallel communication. Nevertheless, parallel communication is desirable due to its low latency and high throughput, which should be supported in `SystemVerilog`. Since the designed has been done in `Verilog` already, this path is thereby not investigated. These above-mentioned points can be the future research work motivated from this project.

References:

[1] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Transaction on Solid-State Circuit,* vol. 52, no. 1, January 2017

[2] V. Dumoulin, and F. Visin, “A guide to convolution arithmetic for deep learning,” available on: https://arxiv.org/abs/1603.07285, March 2016

Appendix I: Python Fixed-point CNN computations

```
# This a script used to parse OPAL generated CNN architecture

import numpy as np
raw_results = np.load("./example_CNN.npz")

# error rate: about 25%
sel = 0

# Define the quantization of the decimal point
f = 8 # a good resolution

# based on the data strructure in the dictionary file:
#        Shape of element 0: (5, 5, 3, 32)
#        Shape of element 1: (32,)
#        Shape of element 2: (32,)
#        Shape of element 3: (32,)
#        Shape of element 4: (32,)
#        Shape of element 5: (32,)
#        Shape of element 6: (3, 3, 32, 64)
#        Shape of element 7: (64,)
#        Shape of element 8: (64,)
#        Shape of element 9: (64,)
#        Shape of element 10: (64,)
#        Shape of element 11: (64,)
#        Shape of element 12: (7, 7, 64, 64)
#        Shape of element 13: (64,)
#        Shape of element 14: (64,)
#        Shape of element 15: (64,)
#        Shape of element 16: (64,)
#        Shape of element 17: (64,)
#        Shape of element 18: (256, 10)
#        Shape of element 19: (10,)
#        Shape of element 20: (10,)
#        Shape of element 21: (10,)
#        Shape of element 22: (10,)
#        Shape of element 23: (10,)

# The first matrix element will be the first layer, consistes of 32 kernels
# kernel_1: 5*5
# second layer: 64 filters
# kernel_2: 3*3
# third layer: 64 filters
# kernel_3: 7*7
# The last layer is fully connected layer

#row_k = [0, 1, 2, 3, 4]                              # row dimension of a
kernel
#col_k = [0, 1, 2, 3, 4]                              # colume dimension of a
kernel
```

```
#cha_k = [0, 1, 2]                                  # channel dimension of a
kernel
#fil_k = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ..., 31] # amount of filter

# The data structure of the CNN layer here is kind of messy,
# it is not the convention way of expression (refer to Eyeriss paper), we
need to reshape it
# kernel =
[[raw_results['pareto_weights'][sel][0][row_k][col_k][cha_k][fil_k]]

fil_k1 = 32
cha_k1 = 3
row_k1 = 5
col_k1 = 5

fil_k2 = 64
cha_k2 = 32
row_k2 = 3
col_k2 = 3

fil_k3 = 64
cha_k3 = 64
row_k3 = 7
col_k3 = 7

# Memory allocations for the targeted reshaped kernels
kernel_1 = np.zeros((fil_k1, cha_k1, row_k1, col_k1))
kernel_2 = np.zeros((fil_k2, cha_k2, row_k2, col_k2))
kernel_3 = np.zeros((fil_k3, cha_k3, row_k3, col_k3))
bias_1 = np.zeros(fil_k1)
bias_2 = np.zeros(fil_k2)
bias_3 = np.zeros(fil_k3)


# reshape the first kernel, and do fixed-point conversion
for u1 in range (0, fil_k1):
    for k1 in range (0, cha_k1):
        for i1 in range (0, row_k1):
            for j1 in range (0, col_k1):
                kernel_1[u1][k1][i1][j1] =
int(raw_results['pareto_weights'][sel][0][i1][j1][k1][u1] * (2**f)) / (2**f)

# reshape the second kernel and do fixed-point conversion
for u2 in range (0, fil_k2):
    for k2 in range (0, cha_k2):
        for i2 in range (0, row_k2):
            for j2 in range (0, col_k2):
                kernel_2[u2][k2][i2][j2] =
int(raw_results['pareto_weights'][sel][6][i2][j2][k2][u2] * (2**f)) / (2**f)

# reshape the third kernel and do fixed-point conversion
for u3 in range (0, fil_k3):
    for k3 in range (0, cha_k3):
```

```
        for i3 in range (0, row_k3):
            for j3 in range (0, col_k3):
                kernel_3[u3][k3][i3][j3] =
int(raw_results['pareto_weights'][sel][12][i3][j3][k3][u3] * (2**f)) / (2**f)

# bias for first layer and do fixed-point conversion
bias_1 = raw_results['pareto_weights'][sel][1]
for p in range (np.size(raw_results['pareto_weights'][0][1])):
    bias_1[p] = int(bias_1[p] * (2**f)) / 2**f

# bias for second layer and do fixed-point conversion
bias_2 = raw_results['pareto_weights'][sel][7]
for q in range (np.size(raw_results['pareto_weights'][0][7])):
    bias_2[q] = int(bias_2[q] * (2**f)) / 2**f

# bias for third layer and do fixed-point conversion
bias_3 = raw_results['pareto_weights'][sel][13]
for r in range (np.size(raw_results['pareto_weights'][0][13])):
    bias_3[q] = int(bias_3[r] * (2**f)) / 2**f

# reshape them to vector
kernel_1_vec = kernel_1.ravel()
kernel_2_vec = kernel_2.ravel()
kernel_3_vec = kernel_3.ravel()
bias_1_vec = bias_1.ravel()
bias_2_vec = bias_2.ravel()
bias_3_vec = bias_3.ravel()

## reshape also testing input to a vector
#I = np.loadtxt("test_batch.txt")
#I_vec = I.ravel()
#
## Save
#np.savetxt('kernel_1_vec.txt', kernel_1_vec, fmt="%.8f")
#np.savetxt('kernel_2_vec.txt', kernel_2_vec, "%.8f")
#np.savetxt('kernel_3_vec.txt', kernel_3_vec, "%.8f")
#np.savetxt('bias_1_vec.txt', bias_1_vec, "%.8f")
#np.savetxt('bias_2_vec.txt', bias_2_vec, "%.8f")
#np.savetxt('bias_3_vec.txt', bias_3_vec, "%.8f")
#np.savetxt('I_vec.txt', I_vec, "%d")

################################################################################
#

# import CIFAR10 input data, 10000 32*32 RGB pics
I = np.loadtxt("test_batch.txt")

# length of the input data
N_1 = np.shape(I)[0]
# Channel dimension
C_1 = 3
# Row dimension of each pic
H_1 = 32
```

```
# Column dimension of each pic
W_1 = 32

# Memory allocation for the reshaped input
I_s = np.zeros((N_1, C_1, H_1, W_1))

# Row counter for I
m = 0
# Column counter for I
n = 0

# Start the reshaping
for zi in range (0, N_1):
    for ki in range (0, C_1):
        for ii in range (0, H_1):
            for ji in range (0, W_1):
                    I_s[zi][ki][ii][ji] = I[m][n]
                    n = n + 1
                    if n == np.shape(I)[1]:
                        n = 0
                        m = m + 1

################################################################################
#

# Start CNN
# Stride for the first layer
U_1 = 1
# Stride for the second layer
U_2 = 2
# Stride for the third layer
U_3 = 3

# First output map row dimension
E_1 = int((H_1 - row_k1 + U_1)/U_1)
# First output map col dimension
F_1 = int((W_1 - col_k1 + U_1)/U_1)

# Second output map row dimension
E_2 = int((E_1 - row_k2 + U_2)/U_2)
# Second output map col dimension
F_2 = int((F_1 - col_k2 + U_2)/U_2)

# Third output map row dimension
E_3 = int((E_2 - row_k3 + U_3)/U_3)
# Second output map col dimension
F_3 = int((F_2 - col_k3 + U_3)/U_3)

# channel dimension of second and third layer
C_2 = 32
C_3 = 64

# Memory allocation for the output from each layer
```

```
O_1 = np.zeros((N_1, fil_k1, E_1, F_1))
O_2 = np.zeros((N_1, fil_k2, E_2, F_2))
O_3 = np.zeros((N_1, fil_k3, E_3, F_3))

# CNN first layer:
# initialize counter in case of counter reusing
z1 = 0
u1 = 0
k1 = 0
i1 = 0
j1 = 0
x1 = 0
y1 = 0
for z1 in range (0, N_1):
    for u1 in range (0, fil_k1):
        for y1 in range (0, E_1):
            for x1 in range (0, F_1):
                for k1 in range (0, C_1):
                    for i1 in range (0, row_k1):
                        for j1 in range (0, col_k1):
                            O_1[z1][u1][y1][x1] = O_1[z1][u1][y1][x1] +
(I_s[z1][k1][U_1*x1+i1][U_1*y1+j1])*(kernel_1[u1][k1][i1][j1])
                            if k1 == (C_1-1):
                                O_1[z1][u1][y1][x1] = O_1[z1][u1][y1][x1] +
bias_1[u1]
                                if O_1[z1][u1][y1][x1] < 0:
                                    O_1[z1][u1][y1][x1] = 0 # ReLU function

# CNN second layer:
# initialize counter in case of counter reusing
z2 = 0
u2 = 0
k2 = 0
i2 = 0
j2 = 0
x2 = 0
y2 = 0
for z2 in range (0, N_1):
    for u2 in range (0, fil_k2):
        for y2 in range (0, E_2):
            for x2 in range (0, F_2):
                for k2 in range (0, C_2):
                    for i2 in range (0, row_k2):
                        for j2 in range (0, col_k2):
                            O_2[z2][u2][y2][x2] = O_2[z2][u2][y2][x2] +
(O_1[z2][k2][U_2*x2+i2][U_2*y2+j2])*(kernel_2[u2][k2][i2][j2])
                            if k2 == (C_2-1):
                                O_2[z2][u2][y2][x2] = O_2[z2][u2][y2][x2] +
bias_2[u2]
                                if O_2[z2][u2][y2][x2] < 0:
                                    O_2[z2][u2][y2][x2] = 0 # ReLU function

# CNN third layer:
```

```
# initialize counter in case of counter reusing
z3 = 0
u3 = 0
k3 = 0
i3 = 0
j3 = 0
x3 = 0
y3 = 0
for z3 in range (0, N_1):
    for u3 in range (0, fil_k3):
        for y2 in range (0, E_3):
            for x3 in range (0, F_3):
                for k3 in range (0, C_3):
                    for i3 in range (0, row_k3):
                        for j3 in range (0, col_k3):
                            O_3[z3][u3][y3][x3] = O_3[z3][u3][y3][x3] +
(O_2[z3][k3][U_3*x3+i3][U_3*y3+j3])*(kernel_3[u3][k3][i3][j3])
                            if k3 == (C_3-1):
                                O_3[z3][u3][y3][x3] = O_3[z3][u3][y3][x3] +
bias_3[u3]
                                if O_3[z3][u3][y3][x3] <0:
                                    O_3[z3][u3][y3][x3] = 0 # ReLU function
```

Appendix II: Matlab Fixed-point decimal to fixed-point binary vector conversion

```
% This program is used to convert input/weight/bias to fixed point binary
% numbers

close all
clear all

kernel_1_vec = importdata('kernel_1_vec.txt'); % fixed-point weight
kernel_2_vec = importdata('kernel_2_vec.txt'); % fixed-point biases
kernel_3_vec = importdata('kernel_3_vec.txt'); % fixed-point weight
bias_1_vec = importdata('bias_1_vec.txt'); % fixed-point biases
bias_2_vec = importdata('bias_2_vec.txt'); % fixed-point biases
bias_3_vec = importdata('bias_3_vec.txt'); % fixed-point biases
I_vec = importdata('I_vec.txt'); % fixed-point biases

% the word-length is 16 bits and 8 bits are for the fraction
length = 16;
frac = 8;

% dimension of each data set
a = size(kernel_1_vec, 1);
b = size(kernel_2_vec, 1);
c = size(kernel_3_vec, 1);
d = size(bias_1_vec, 1);
e = size(bias_2_vec, 1);
f = size(bias_3_vec, 1);
g = size(I_vec, 1);

% convert the first weight vector to binary fixed point
for k = 1:a
    if kernel_1_vec(k, 1) >= 0
        kernel_1_b(k, 1) = string(dec2bin(floor(2^frac * kernel_1_vec(k, 1)),
length));
    else
        kernel_1_b(k, 1) = string(dec2bin(floor(2^length + 2^frac *
kernel_1_vec(k, 1)), length));
    end
end

% convert the second weight vector to binary fixed point
for l = 1:b
    if kernel_2_vec(l, 1) >= 0
        kernel_2_b(l, 1) = string(dec2bin(floor(2^frac * kernel_2_vec(l, 1)),
length));
    else
        kernel_2_b(l, 1) = string(dec2bin(floor(2^length + 2^frac *
kernel_2_vec(l, 1)), length));
    end
end

% convert the third weight vector to binary fixed point
for m = 1:c
    if kernel_3_vec(m, 1) >= 0
```

```
        kernel_3_b(m, 1) = string(dec2bin(floor(2^frac * kernel_3_vec(m, 1)),
length));
    else
        kernel_3_b(m, 1) = string(dec2bin(floor(2^length + 2^frac *
kernel_3_vec(m, 1)), length));
    end
end

% convert the first bias vector to binary fixed point
for n = 1:d
    if bias_1_vec(n, 1) >= 0
        bias_1_b(n, 1) = string(dec2bin(floor(2^frac * bias_1_vec(n, 1)),
length));
    else
        bias_1_b(n, 1) = string(dec2bin(floor(2^length + 2^frac *
bias_1_vec(n, 1)), length));
    end
end

% convert the second bias vector to binary fixed point
for o = 1:e
    if bias_2_vec(o, 1) >= 0
        bias_2_b(o, 1) = string(dec2bin(floor(2^frac * bias_2_vec(o, 1)),
length));
    else
        bias_2_b(o, 1) = string(dec2bin(floor(2^length + 2^frac *
bias_2_vec(o, 1)), length));
    end
end

% convert the third bias vector to binary fixed point
for p = 1:f
    if bias_3_vec(p, 1) >= 0
        bias_3_b(p, 1) = string(dec2bin(floor(2^frac * bias_3_vec(p, 1)),
length));
    else
        bias_3_b(p, 1) = string(dec2bin(floor(2^length + 2^frac *
bias_3_vec(p, 1)), length));
    end
end

% convert the input vector to binary fixed point
for q = 1:g
    if I_vec(q, 1) >= 0
        I_b(q, 1) = string(dec2bin(floor(2^frac * I_vec(q, 1)), length));
    else
        I_b(q, 1) = string(dec2bin(floor(2^length + 2^frac * I_vec(q, 1)),
length));
    end
end

% Save them to txt file
kernel_1_b_ID = fopen('kernel_1_b.txt','w');
input0 = kernel_1_b;
fprintf(kernel_1_b_ID, '%s\n', input0{:});
fclose(kernel_1_b_ID);
```

```
kernel_2_b_ID = fopen('kernel_2_b.txt','w');
input1 = kernel_2_b;
fprintf(kernel_2_b_ID, '%s\n', input1{:});
fclose(kernel_2_b_ID);

kernel_3_b_ID = fopen('kernel_3_b.txt','w');
input2 = kernel_3_b;
fprintf(kernel_3_b_ID, '%s\n', input2{:});
fclose(kernel_3_b_ID);

bias_1_b_ID = fopen('bias_1_b.txt','w');
input3 = bias_1_b;
fprintf(bias_1_b_ID, '%s\n', input3{:});
fclose(bias_1_b_ID);

bias_2_b_ID = fopen('bias_2_b.txt','w');
input4 = bias_2_b;
fprintf(bias_2_b_ID, '%s\n', input4{:});
fclose(bias_2_b_ID);

bias_3_b_ID = fopen('bias_3_b.txt','w');
input5 = bias_3_b;
fprintf(bias_3_b_ID, '%s\n', input5{:});
fclose(bias_3_b_ID);

I_b_ID = fopen('I_b.txt','w');
input6 = I_b;
fprintf(I_b_ID, '%s\n', input6{:});
fclose(I_b_ID);
```

Appendix III: HDL implementation of CNN convolutional accelerator

```
// This script implement first layer CNN convolutional computation
// Using 16-bit fixed point calculation
// Classification rate is smaller than 28%
// Input data sets are 10000 32*32 RGB CIFAR10 pictures

// Generated by OPAL:
/* # based on the data strructure in the dictionary file:
#        Shape of element 0: (5, 5, 3, 32)
#        Shape of element 1: (32,)
#        Shape of element 2: (32,)
#        Shape of element 3: (32,)
#        Shape of element 4: (32,)
#        Shape of element 5: (32,)
#        Shape of element 6: (3, 3, 32, 64)
#        Shape of element 7: (64,)
#        Shape of element 8: (64,)
#        Shape of element 9: (64,)
#        Shape of element 10: (64,)
#        Shape of element 11: (64,)
#        Shape of element 12: (7, 7, 64, 64)
#        Shape of element 13: (64,)
#        Shape of element 14: (64,)
#        Shape of element 15: (64,)
#        Shape of element 16: (64,)
#        Shape of element 17: (64,)
#        Shape of element 18: (256, 10)
#        Shape of element 19: (10,)
#        Shape of element 20: (10,)
#        Shape of element 21: (10,)
#        Shape of element 22: (10,)
#        Shape of element 23: (10,)

# The first matrix element will be the first layer, consistes of 32 kernels
# kernel_1: 5*5
# second layer: 64 filters
# kernel_2: 3*3
# third layer: 64 filters
# kernel_3: 7*7
# The last layer is fully connected layer */
```

```
module MAC (IN1, IN2, CLK, CLR_MAC, CLR_ACC, OUT_MAC);
//////////////////////////////////////////////////////////
      // Declare paremeters
      parameter                                          INT = 8;
            // integer resolution of the input/output
      parameter                                          DEC = 8;
            // decimal resolution of the input/output

      input signed [INT-1:-DEC]                    IN1;
```

```
      input signed [INT-1:-DEC]                         IN2;
      input                                                   CLK,
CLR_MAC, CLR_ACC; // CLR_MAC: clear signal for MAC; CLR_ACC: clear signal for
accumulator
      output reg signed [INT-1:-DEC]                    OUT_MAC;

      // Declare registers and wires
      reg signed [INT-1:-DEC]                           IN1_reg, IN2_reg;
      reg                                                     CLR_ACC_reg;
      reg signed [INT-1:-DEC]                           ACCOut;
      reg signed [2*INT-1:-2*DEC]                       MUL;
      wire signed [INT-1:-DEC]                          MULOut;
            // trucated version of MUL
//////////////////////////////////////////////////////////

      assign      MULOut = MUL[INT-1:-DEC];

      // Fixed point multiplications
      always @ (posedge CLK) begin
            // Store the results of the operations on the current data
            if (IN1[INT-1] == 0 & IN2[INT-1] == 0) begin
                  MUL <= IN1 * IN2;
            end
            // signed by unsigned (negative by positive)
            if (IN1[INT-1] == 1 & IN2[INT-1] == 0) begin
                  MUL <= (
                              {{16{IN2[-DEC]}}, (IN1 * IN2[-DEC])} +
                              {{15{IN2[-DEC+1]}}, (IN1 * IN2[-DEC+1])<<1} +
                              {{14{IN2[-DEC+2]}}, (IN1 * IN2[-DEC+2])<<2} +
                              {{13{IN2[-DEC+3]}}, (IN1 * IN2[-DEC+3])<<3} +
                              {{12{IN2[-DEC+4]}}, (IN1 * IN2[-DEC+4])<<4} +
                              {{11{IN2[-DEC+5]}}, (IN1 * IN2[-DEC+5])<<5} +
                              {{10{IN2[-DEC+6]}}, (IN1 * IN2[-DEC+6])<<6} +
                              {{9{IN2[-DEC+7]}}, (IN1 * IN2[-DEC+7])<<7} +
                              {{8{IN2[-DEC+8]}}, (IN1 * IN2[-DEC+8])<<8} +
                              {{7{IN2[-DEC+9]}}, (IN1 * IN2[-DEC+9])<<9} +
                              {{6{IN2[-DEC+10]}}, (IN1 * IN2[-DEC+10])<<10} +
                              {{5{IN2[-DEC+11]}}, (IN1 * IN2[-DEC+11])<<11} +
                              {{4{IN2[-DEC+12]}}, (IN1 * IN2[-DEC+12])<<12} +
                              {{3{IN2[-DEC+13]}}, (IN1 * IN2[-DEC+13])<<13} +
                              {{2{IN2[-DEC+14]}}, (IN1 * IN2[-DEC+14])<<14} +
                              {{1{IN2[-DEC+15]}}, (IN1 * IN2[-DEC+15])<<15}
                              );
            end
            // unsigned by signed (positive by negative)
            if (IN1[INT-1] == 0 & IN2[INT-1] == 1) begin
                  MUL <= (
                              (IN1 * IN2[-DEC]) +
                              ((IN1 * IN2[-DEC+1])<<1) +
                              ((IN1 * IN2[-DEC+2])<<2) +
                              ((IN1 * IN2[-DEC+3])<<3) +
                              ((IN1 * IN2[-DEC+4])<<4) +
                              ((IN1 * IN2[-DEC+5])<<5) +
                              ((IN1 * IN2[-DEC+6])<<6) +
                              ((IN1 * IN2[-DEC+7])<<7) +
                              ((IN1 * IN2[-DEC+8])<<8) +
                              ((IN1 * IN2[-DEC+9])<<9) +
```

```
                              ((IN1 * IN2[-DEC+10])<<10) +
                              ((IN1 * IN2[-DEC+11])<<11) +
                              ((IN1 * IN2[-DEC+12])<<12) +
                              ((IN1 * IN2[-DEC+13])<<13) +
                              ((IN1 * IN2[-DEC+14])<<14) +
                              ((1'b1 + ~{1'b0, (IN1 * IN2[-DEC+15])})<<15)
      // last number should add a positive '0' at the MSB and then take the
2's complement
                              );
            end

            // signed by signed (negative by negative)
            if (IN1[INT-1] == 1 & IN2[INT-1] == 1) begin
                  MUL <= ((~IN1 + 1'b1) * (~IN2 + 1'b1)) << 1;
      // drop the redundant sign bit by left shifting, "7|9"
            end
      end

      // Store the value of the accumulation (or clear it)
      always @ (OUT_MAC, CLR_ACC_reg)
      begin
            if (CLR_ACC_reg)
                  ACCOut <= 0;
            else
                  ACCOut <= OUT_MAC;
      end

      // Clear or update data, as appropriate
      always @ (posedge CLK or posedge CLR_MAC)
      begin
            if (CLR_MAC)
            begin
                  IN1_reg <= 0;
                  IN2_reg <= 0;
                  CLR_ACC_reg <= 0;
                  OUT_MAC <= 0;
            end

            else
            begin
                  IN1_reg <= IN1;
                  IN2_reg <= IN2;
                  CLR_ACC_reg <= CLR_ACC;
                  if (IN1[INT-1] == 1 & IN2[INT-1] == 1) begin
                        OUT_MAC <= ACCOut + (MULOut >> 1);
            // shift the 'MUL' back after dropping the redundant bit, "7|9 ->
8|8"
                  end
                  else begin
                        OUT_MAC <= ACCOut + MULOut;
            // DONT shift the if it is not 'signed * signed', "8|8"
                  end
            end
      end
endmodule
```

```
// Load the kenel 1 "K1" to BRAM
module BRAM_K1 (Pointer_K1, K1);
////////////////////////////////////////////////////////
      parameter                                       INT = 8;
      // integer resolution of the input/output
      parameter                                       DEC = 8;
      // decimal resolution of the input/output

      input [11:0]                                    Pointer_K1;
      // index for first K1 vector, 2400 elements
      output reg signed [INT-1:-DEC]            K1;
      // signed binary output
      reg [INT-1:-DEC]                          BRAM_K1 [0:2400-1];
////////////////////////////////////////////////////////

      always @ (Pointer_K1) begin
            K1 = BRAM_K1 [Pointer_K1];
      end

      initial begin
            $readmemb("kernel_1_b.txt", BRAM_K1);
      end
endmodule




// Load the bias 1 "B1" to BRAM
module BRAM_B1 (Pointer_B1, B1);
////////////////////////////////////////////////////////
      parameter                                       INT = 8;
      // integer resolution of the input/output
      parameter                                       DEC = 8;
      // decimal resolution of the input/output

      input [4:0]                                     Pointer_B1;
      // index for first B1 vector, 32 elements
      output reg signed [INT-1:-DEC]            B1;
      // signed binary output
      reg [INT-1:-DEC]                          BRAM_B1 [0:32-1];
////////////////////////////////////////////////////////

      always @ (Pointer_B1) begin
            B1 = BRAM_B1 [Pointer_B1];
      end

      initial begin
            $readmemb("bias_1_b.txt", BRAM_B1);
      end
endmodule
```

```
// Load the all input feature map to BRAM, this is just for testing
module BRAM_IF (Pointer_IF, IF);
//////////////////////////////////////////////////////////
      parameter                                       INT = 8;
            // integer resolution of the input/output
      parameter                                       DEC = 8;
            // decimal resolution of the input/output

      input [13:0]                                    Pointer_IF;
            // index for first B1 vector, 10000*(32*32*3) = 30720000 elements
      output reg signed [INT-1:-DEC]            IF;
            // signed binary output
      reg [INT-1:-DEC]                          BRAM_IF [0:30720-1];
      // say test the first 10 pics
//////////////////////////////////////////////////////////

      always @ (Pointer_IF) begin
            IF = BRAM_IF [Pointer_IF];
      end

      initial begin
            $readmemb("I_b.txt", BRAM_IF);
      end
endmodule




// Processing element
module PE(CLK, RST, Pointer_K1, Pointer_IF, Addr_PE, ACK, OUT_PE);
//////////////////////////////////////////////////////////
      parameter                                       INT = 8;
                                    // integer resolution of the input/output
      parameter                                       DEC = 8;
                                    // decimal resolution of the input/output
      parameter                                       E = 28;
                                          // row dimension of the output
feature maps
      parameter                                       R = 5;
                                          // row dimension of the filter
      parameter                                       U = 1;
                                          // stride

      input                                                 CLK, RST;
      input [11:0]                                    Pointer_K1;
      input [13:0]                                    Pointer_IF;
                                    // pointers for kernel_1 and input
feature maps
      input [4:0]                                     Addr_PE;
                                    // accessing address (0-27) of the
"OUT_PE"
```

```
      output signed [INT-1:-DEC]                OUT_PE;
                              // wiring the OF_reg to output; Verilog does
not support parallel output, SystemVerilog does
      output                                                  ACK;

      reg signed [INT-1:-DEC]                   OF_reg [0:E-1];
                              // output reg from one PE
      reg                                                     ACK_reg;
                                    // ackowledgement signal that means one
row is finished
      reg [2:0]                                               Counter_5;
                                    // count up to 5 (0-5), if it is 5, the
row of input feature map shifts rights (0 is for skipping the initial
condition)
      reg [4:0]                                               Counter_28;
                                    // count up to 28 (0-28), if it is 28,
shift to another channel (both kernel and ifmap)
      reg [4:0]                                               Buff_Counter;
                                    // buffering for the PE, tweak this guy
to obtain more delays in between PEs, so that psum results wont be corrupted

      reg [11:0]                                              Pointer_K1_reg;
      reg [13:0]                                              Pointer_IF_reg;
                                    // pointers for kernel_1 and input
feature maps

      reg                                                     CLR_MAC, CLR_ACC;


      wire signed [INT-1:-DEC]                  K1, IF;
      wire signed [INT-1:-DEC]                  OUT_MAC;
////////////////////////////////////////////////////////////

      BRAM_K1 Kernel_1(Pointer_K1_reg, K1);
      BRAM_IF IF_Maps(Pointer_IF_reg, IF);
      MAC MAC_1(K1, IF, CLK, CLR_MAC, CLR_ACC, OUT_MAC);

      assign OUT_PE = OF_reg[Addr_PE];

      assign ACK = ACK_reg;

      always @ (posedge CLK) begin
            if (RST || ACK_reg)
                  // reset everything
            begin
                  Counter_5 <= 0;
                  Counter_28 <= 0;
                  Buff_Counter <= 0;
                  ACK_reg <= 0;
                  CLR_MAC <= 1;
                  CLR_ACC <= 1;
                  Pointer_K1_reg <= 0;
                  Pointer_IF_reg <= 0;
            end

            else if (Counter_5 == 0)
                  // a buffer state that is ued to do the initialization
```

```
begin
      if (Counter_28 == 0)
      begin
            if (Buff_Counter < E)
            begin
                  Buff_Counter <= Buff_Counter + 1;
                  Counter_28 <= 0;
      // force it back to start this if loop again
            end
            else begin
                  CLR_MAC <= 0;
                  CLR_ACC <= 0;
                  Pointer_K1_reg <= Pointer_K1;
// load new kernel row data to the PE at initial condition
                  Pointer_IF_reg <= Pointer_IF;
// load new ifmap row data to the PE at initial condition
                  Counter_5 <= Counter_5 + 1;
            end
      end
end

else if (Counter_5 == 1)
      // actual computation starts from here
begin
      CLR_ACC <= 0;
      Pointer_K1_reg <= Pointer_K1_reg + 1;
      Pointer_IF_reg <= Pointer_IF_reg + 1;
      Counter_5 <= Counter_5 + 1;
      OF_reg[Counter_28-1] <= OUT_MAC;
      if (Counter_28 == 28)
      // reset the ifmap pointer to 0 if it is 28
      begin
            ACK_reg <= 1;
      end
end

else if (Counter_5 == 2)
begin
      CLR_ACC <= 0;
      Pointer_K1_reg <= Pointer_K1_reg + 1;
// increment pointer of the kernel
      Pointer_IF_reg <= Pointer_IF_reg + 1;
// increment pointer of the ifmap
      Counter_5 <= Counter_5 + 1;
      // goes to next state
end

else if (Counter_5 == 3)
      // sames as above
begin
      CLR_ACC <= 0;
      Pointer_K1_reg <= Pointer_K1_reg + 1;
      Pointer_IF_reg <= Pointer_IF_reg + 1;
      Counter_5 <= Counter_5 + 1;
end
```

```
            else if (Counter_5 == 4)
                  // same as above
            begin
                  CLR_ACC <= 0;
                  Pointer_K1_reg <= Pointer_K1_reg + 1;
                  Pointer_IF_reg <= Pointer_IF_reg + 1;
                  Counter_5 <= Counter_5 + 1;
            end

            else if (Counter_5 == 5)
                  // transient state
            begin
                  CLR_ACC <= 1;
                        // clear the accumulation of the MAC
                  Counter_5 <= 1;
                        // back to state where "Counter_5 = 1"
                  Pointer_K1_reg <= Pointer_K1_reg - (R - 1);
      // shift the filter back to original positions
                  Pointer_IF_reg <= Pointer_IF_reg - (R - U - 1);
      // shift ifmap row by "U"
                  Counter_28 <= Counter_28 + 1;
            end
      end

endmodule




// first CNN layer implementation; 5*28 PEs
module CNN (CLK, RST);
//////////////////////////////////////////////////////////////
      parameter                                       INT = 8;

            // integer resolution of the input/output
      parameter                                       DEC = 8;

            // decimal resolution of the input/output
      parameter                                       C = 3;

                  // channel dimension of ifmap and kernel
      parameter                                       E = 28;

                  // row dimension of the ofmap
      parameter                                       F = 28;

                  // column dimension of the ofmap
      parameter                                       R = 5;

                  // row dimension of the filter
      parameter                                       H = 32;

                  // row dimension of the ifmap
      parameter                                       M = 32;
```

```
                // number of filters

      input                                                CLK, RST;


      reg [11:0]                                           Pointer_K1 [0:(R * E -
1)];
      // pointer for each row of the kernel 1
      reg [11:0]                                           Pointer_K1_psum   [0:(R
- 1)];
      // group each PEs for one filter row pointer
      reg [13:0]                                           Pointer_IF [0:(R * E -
1)];
      // pointer for each row of the ifmap
      reg [13:0]                                           Pointer_IF_psum   [0:(H
- 1)];
      // pointer for each diagnoal of PE arrays.
      reg [4:0]                                            Addr_PE [0:(R * E -
1)];
      // address for retriving output from each PE

      reg [4:0]                                            Addr_psum;

            // address for controlling all PEs

      reg [2:0]                                            Counter_C;

            // channel counter, counts from 0-2
      reg [4:0]                                            Counter_K;

            // kernel counter, counts from 0-31 (also the address of bias)
      reg [13:0]                                           Counter_IF;

            // ifmap counter, in this example, it countes 0-9 10 pics
      reg                                                  Switch_psum;

            // swwitch for the psum result adding
      reg                                                  ACK_ofmap;

            // ackowledgement signal to tell one ofmap is completed
      wire signed [INT-1:-DEC]                  Bias;

      // bias for the first layer


      wire                                                 ACK_PE [0:(R * E - 1)];

      // ackowledgement signal from each PE
      wire                                                 ACK_psum;

            // synchrounous ACK signal

      wire signed [INT-1:-DEC]                  OUT_PE [0:(R * E - 1)];

      // output wire from each PE
```

```
      reg signed [INT-1:-DEC]                   OUT_psum [0:(E * F * M - 1)];
                                                                              //
psum from all columes of PEs, ofmap (32 channels, each channel has 28*28)

////////////////////////////////////////////////////////////

      genvar a;
      generate
          for (a = 0; a < (E * R); a = a + 1) begin: pe_array
                                                                              //
generate 5*28 PEs
                  PE PE_Array(CLK, RST, Pointer_K1[a], Pointer_IF[a],
Addr_PE[a], ACK_PE[a], OUT_PE[a]);
            end
      endgenerate

      BRAM_B1 BIAS(Counter_K, Bias);
                                                // loading bias

      // initilization...tie different registers together...
      integer m;
                                                        // for Pointer_K1
initialization

      integer n;
                                                        // for Pointer_IF
initialization

      integer i;
                                                        // for Pointer_K1 to
Pointer_K1_psum mapping
      integer j;
                                                        // for Pointer_K1 to
Pointer_K1_psum mapping

      integer k;
                                                        // for pointer_IF to
pointer_IF_psum
      integer l;
                                                        // for pointer_IF to
pointer_IF_psum

      integer p;
                                                        // for Addr_PE to
Addr_psum mapping
      integer q;
                                                        // for Addr_PE to
Addr_psum mapping
      integer w;
                                                        // for Addr_psum
initilization

      integer u;
                                                        // for OUT_psum
initilization
```

```
      integer kernel_p0;
                                                      // kernel pointer
      integer ifmap_p0;
                                                // ifmap pointer

      integer kernel_p1;
                                                      // kernel pointer
      integer ifmap_p1;
                                                // ifmap pointer

      integer kernel_p2;
                                                      // reset the kernel
      integer ifmap_p2;
                                                // ifmap pointer

      integer kernel_p3;
                                                      // kernel pointer
      integer ifmap_p3;
                                                // ifmap pointer

      integer kernel_p4;
                                                      // kernel pointer
      integer ifmap_p4;
                                                // ifmap pointer

      integer x;
                                                      // psum counter
      integer y;
                                                      // psum counter

      genvar r;
                                                      // for wire ACK_PE to
ACK_psum mapping
      generate
      for (r = 0; r < (R * E); r = r + 1)
      begin: ack_assign
            assign ACK_psum = ACK_PE[r];
                                    // tie all wires ACK_PE with just one
ACK_psum signal as they are all synchrounous
      end
      endgenerate

      always @ (posedge CLK) begin
            if (RST)
                                                      // State -1: reset
everything back to initial condition (initialization stage)
            begin
                  Counter_C <= 0;
                                                      // channel counter
                  Counter_K <= 0;
                                                      // kernel counter
                  Counter_IF <= 0;
                                                // ifmap counter
                  Switch_psum <= 0;
                                                // psum switch initilization
                  for (m = 0; m < R; m = m + 1)
                  begin
```

```
                        Pointer_K1_psum[m] <= 0;
                                                // initialize Pointer_K1_psum
                  end

                  for (n = 0; n < M; n = n + 1)
                  begin
                        Pointer_IF_psum[n] <= 0;
                                                // initialize Pointer_IF_psum
                  end

                  Addr_psum <= 0;
                                                      // initialize Addr_psum

                  for (u = 0; u < (E * F * M); u = u + 1)
                  begin
                        OUT_psum[u] <= 0;
                                                // initialize OUT_psum
                  end

            end

            else if (ACK_psum == 0 && Counter_C == 0 && Counter_K == 0)
                              // State 0: initial state
            begin
                  for (kernel_p0 = 0; kernel_p0 < R; kernel_p0 = kernel_p0 +
1)
                  begin
                        Pointer_K1_psum[kernel_p0] <= kernel_p0 * R;
                  end
                  for (ifmap_p0 = 0; ifmap_p0 < H; ifmap_p0 = ifmap_p0 + 1)
                  begin
                        Pointer_IF_psum[ifmap_p0] <= ifmap_p0 * H;
                  end
            end

            else if (ACK_psum == 0)
                                                // State 1: when there is not
ACK_psum signal, they remain themselves
            begin
                  for (kernel_p1 = 0; kernel_p1 < R; kernel_p1 = kernel_p1 +
1)
                  begin
                        Pointer_K1_psum[kernel_p1] <=
Pointer_K1_psum[kernel_p1];
                  end
                  for (ifmap_p1 = 0; ifmap_p1 < H; ifmap_p1 = ifmap_p1 + 1)
                  begin
                        Pointer_IF_psum[ifmap_p1] <=
Pointer_IF_psum[ifmap_p1];
                  end
            end

            // state 2.0 and state 2.1 are NOT exclusive
            if (ACK_psum == 1)
                                                      // State 2.0: as long
as there is an ACK_psum == 1, increment the channel counter no matter what
            begin
```

```
                  Switch_psum <= 1;
                                                // turn on the switch the
adding of the results
                  if (Counter_C == (C - 1))
                                                // if all 3 channels of one
kernel are finished
                  begin
                        Counter_C <= 0;
                                                // means one kernel is
completed
                        if (Counter_K == (M - 1))
                                                // if all kernels are used
for one ifmap
                        begin
                              Counter_K <= 0;
                                                      // reset the kernel
pointer back to initial
                              for (kernel_p2 = 0; kernel_p2 < R; kernel_p2 =
kernel_p2 + 1)
                              begin
                                    Pointer_K1_psum[kernel_p2] <= kernel_p2 *
R;
                              end
                              for (ifmap_p2 = 0; ifmap_p2 < H; ifmap_p2 =
ifmap_p2 + 1)
                              begin
                                    Pointer_IF_psum[ifmap_p2] <=
Pointer_IF_psum[ifmap_p2] + (H * H);      // ifmap shifts one channel, a new
ifmap comes in
                              end
                              Counter_IF <= Counter_IF + 1;
                                          // increment ifmap counter
                        end

                        else
                        begin
                              for (kernel_p3 = 0; kernel_p3 < R; kernel_p3 =
kernel_p3 + 1)
                              begin
                                    Pointer_K1_psum[kernel_p3] <=
Pointer_K1_psum[kernel_p3] + (R * R);     // kernel shifts one channel
                              end

                              for (ifmap_p3 = 0; ifmap_p3 < H; ifmap_p3 =
ifmap_p3 + 1)
                              begin
                                    Pointer_IF_psum[ifmap_p3] <=
Pointer_IF_psum[ifmap_p3] + (H * H);      // ifmap shifts one channel
                              end
                        end
                        Counter_K <= Counter_K + 1;
                                          // shift to next kernel
                  end

                  else
                  begin
```

```
                        for (kernel_p4 = 0; kernel_p4 < R; kernel_p4 =
kernel_p4 + 1)
                        begin
                              Pointer_K1_psum[kernel_p4] <=
Pointer_K1_psum[kernel_p4] + (R * R);           // kernel shifts one channel
                        end

                        for (ifmap_p4 = 0; ifmap_p4 < H; ifmap_p4 = ifmap_p4
+ 1)
                        begin
                              Pointer_IF_psum[ifmap_p4] <=
Pointer_IF_psum[ifmap_p4] + (H * H);            // ifmap shifts one channel
                        end
                        Counter_C <= Counter_C + 1;
                  end
            end

            if (Switch_psum == 1)
                                                // State 2.1: psum
calculation
            begin
                  if (Addr_psum < E)
                  begin
                        if (Counter_C < (C - 1))
                        begin
                              for (x = 0; x < E; x = x + 1)
                              begin
                                    OUT_psum[Addr_psum + E * x + E * F *
Counter_K] <= OUT_PE[x * 5]
                                                      + OUT_PE[x * 5 + 1]
                                                      + OUT_PE[x * 5 + 2]
                                                      + OUT_PE[x * 5 + 3]
                                                      + OUT_PE[x * 5 + 4]
                                                      + OUT_psum[Addr_psum +
E * x + E * F * Counter_K];         // adding psum with the convolution
results from all channels and allocate them to different channel based on

                                                                  // which kernel
is the input
                              end
                        end
                        else
                        begin
                              for (y = 0; y < E; y = y + 1)
                              begin
                                    OUT_psum[Addr_psum + E * y + E * F *
Counter_K] <= OUT_PE[y * 5]
                                                      + OUT_PE[y * 5 + 1]
                                                      + OUT_PE[y * 5 + 2]
                                                      + OUT_PE[y * 5 + 3]
                                                      + OUT_PE[y * 5 + 4]
                                                      + OUT_psum[Addr_psum +
E * y + E * F * Counter_K]
                                                      + Bias;
                                                            // adding the
result to the bias if one kernel is finished
```

```
                                    if (OUT_psum[Addr_psum + E * y + E * F *
Counter_K] < 0)
                                    begin
                                          OUT_psum[Addr_psum + E * y + E * F
* Counter_K] <= 0;                   // ReLU function
                                    end
                                    else
                                    begin
                                          OUT_psum[Addr_psum + E * y + E * F
* Counter_K]
                                          <= OUT_psum[Addr_psum + E * y + E *
F * Counter_K];
                                    end
                              end
                        end

                        if (Counter_C == (C - 1) && Counter_K == (M - 1))
                                    // if all the convolutions in between
kernel sets and one ifmap (all channels) are finished
                        begin
                              ACK_ofmap <= 1;
                                                      // send an
ackowledgement signal to tell one ofmap is finished and data are ready for
accessed
                        end
                        else
                        begin
                              ACK_ofmap <= 0;
                        end

                        if (Addr_psum !== (E - 1))
                        begin
                              Addr_psum <= Addr_psum + 1;
                        end
                        else
                        begin
                              Addr_psum <= 0;
                              Switch_psum <= 0;
                        end
                  end
            end
      end

      always @ (*) begin
                                                      // State0: mapping the
PEs with appropriate pointers
            for (i = 0; i < R; i = i + 1)
            begin
                  for (j = 0; j < (E * R); j = j + 5)
                  begin
                        Pointer_K1[i + j] <= Pointer_K1_psum[i];
                                    // tie PEs with same kernel pointers,
CHECKED!
                  end
            end

            for (p = 0; p < (E * R); p = p + 1)
```

```
            begin
                  Addr_PE[p] <= Addr_psum;
                                                // use one address signal to
control all PEs, CHECKED!
            end

            //-------------------------------------//
                                    // tricky... map Pointer_IF to
Pointer_psum diagnoally
            Pointer_IF[0] <= Pointer_IF_psum[0];
                                          // maps ifmap row 1 to
Pointer_IF_psum diagnoally

            Pointer_IF[1] <= Pointer_IF_psum[1];
                                          // maps ifmap row 2 to
Pointer_IF_psum diagnoally
            Pointer_IF[5] <= Pointer_IF_psum[1];

            Pointer_IF[2] <= Pointer_IF_psum[2];
                                          // maps ifmap row 3 to
Pointer_IF_psum diagnoally
            Pointer_IF[6] <= Pointer_IF_psum[2];
            Pointer_IF[10] <= Pointer_IF_psum[2];

            Pointer_IF[3] <= Pointer_IF_psum[3];
            Pointer_IF[7] <= Pointer_IF_psum[3];
            Pointer_IF[11] <= Pointer_IF_psum[3];
            Pointer_IF[15] <= Pointer_IF_psum[3];

            for (k = 4; k < 28; k = k + 1)
                                                // maps ifmap row 4-27 to
Pointer_IF_psum diagnoally
            begin
                  for (l = 0; l < R; l = l + 1)
                  begin
                        Pointer_IF[k + l * 4 + 4 * (k - 4)] <=
Pointer_IF_psum[k];                             // "k + l * 4 + 4 * (k - 4)
<= k", very tricky correlation...., CHECKED!
                  end
            end
            Pointer_IF[124] <= Pointer_IF_psum[28];
                                          // maps ifmap row 28 to
Pointer_IF_psum diagnoally
            Pointer_IF[128] <= Pointer_IF_psum[28];
            Pointer_IF[132] <= Pointer_IF_psum[28];
            Pointer_IF[136] <= Pointer_IF_psum[28];

            Pointer_IF[129] <= Pointer_IF_psum[29];
                                          // maps ifmap row 29 to
Pointer_IF_psum diagnoally
            Pointer_IF[133] <= Pointer_IF_psum[29];
            Pointer_IF[137] <= Pointer_IF_psum[29];

            Pointer_IF[134] <= Pointer_IF_psum[30];
                                          // maps ifmap row 30 to
Pointer_IF_psum diagnoally
            Pointer_IF[138] <= Pointer_IF_psum[30];
```

```
            Pointer_IF[139] <= Pointer_IF_psum[31];
                                                // maps ifmap row 31 to
Pointer_IF_psum diagnoally
            //-------------------------------------//
      end

endmodule
```

Appendix IV: Xilinx Viavado 2014.3.1 synthesis report on utilization (Virtex-7 xc7vx485tffg1157-1)


```
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2014.3.1 (win64) Build 1056140 Thu Oct 30 17:03:40
MDT 2014
| Date         : Sat Dec 16 21:05:02 2017
| Host         : 156Pavel-Sinha running 64-bit Service Pack 1  (build 7601)
| Command      : report_utilization -file CNN_utilization_synth.rpt -pb
CNN_utilization_synth.pb
| Design       : CNN
| Device       : xc7vx485t
| Design State : Synthesized
------------------------------------------------------------------------------------
```

Utilization Design Information

Table of Contents

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2660	0	303600	0.87
LUT as Logic	2660	0	303600	0.87
LUT as Memory	0	0	130800	0.00
Slice Registers	1960	0	607200	0.32
Register as Flip Flop	1960	0	607200	0.32
Register as Latch	0	0	607200	0.00
F7 Muxes	0	0	151800	0.00
F8 Muxes	0	0	75900	0.00

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
0	Yes	-	Reset
140	Yes	Set	-
1820	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	1030	0.00
RAMB36/FIFO*	0	0	1030	0.00
RAMB18	0	0	2060	0.00

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	0	0	2800	0.00

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	2	0	600	0.33
Bonded IPADs	0	0	62	0.00
Bonded OPADs	0	0	40	0.00
PHY_CONTROL	0	0	14	0.00
PHASER_REF	0	0	14	0.00
OUT_FIFO	0	0	56	0.00
IN_FIFO	0	0	56	0.00
IDELAYCTRL	0	0	14	0.00
IBUFGDS	0	0	576	0.00
GTXE2_COMMON	0	0	5	0.00

GTXE2_CHANNEL	0	0	20	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	56	0.00
PHASER_IN/PHASER_IN_PHY	0	0	56	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	700	0.00
ODELAYE2/ODELAYE2_FINEDELAY	0	0	700	0.00
IBUFDS_GTE2	0	0	28	0.00
ILOGIC	0	0	600	0.00
OLOGIC	0	0	600	0.00

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.12
BUFIO	0	0	56	0.00
MMCME2_ADV	0	0	14	0.00
PLLE2_ADV	0	0	14	0.00
BUFMRCE	0	0	28	0.00
BUFHCE	0	0	168	0.00
BUFR	0	0	56	0.00

6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
PCIE_2_1	0	0	4	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
FDRE	1820	Flop & Latch
LUT5	700	LUT
LUT4	700	LUT
LUT3	700	LUT
LUT2	560	LUT
LUT6	420	LUT

LUT1	140	LUT
FDSE	140	Flop & Latch
IBUF	2	IO
BUFG	1	Clock

8. Black Boxes

Ref Name	Used

9. Instantiated Netlists

Ref Name	Used