ECE 284 PROJECT REPORT

GROUPNAME : ACCELEXPRO

Implementation of VGG16 and ResNet
on 2D Systolic Array and FPGA Mapping

Group Members

Atul Acharya (A59004891)
(aacharya@ucsd.edu)

Kanishka Sharma(A59005455)
(k2sharma@ucsd.edu)

Shreyas Borse(A59009564)
(sborse@ucsd.edu)

Akshay Joshi (A59005428)
(akjoshi@ucsd.edu)

\

Linux Cloud Path: /home/linux/ieng6/ee284fa22/sborse/ece284fa22/2D-Systolic-array-processor-ISA

Github Link: https://github.com/Shreyas-borse/2D-Systolic-array-processor-ISA

Part-1: Train VGG16 and RESNET with quantization-aware training

VGG16 :- The 27th layer is modified to have 8 input channels and 8 output channels and the batch normalization layer is removed.
The **accuracy** achieved is **92%** after quantization-aware training and the **psum_recovered error** is **-2.9*e-8**

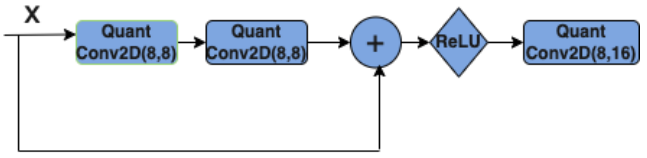
```
(27): QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
(28): ReLU(inplace=True)
```

RESNET20 :- The layer1(0) is modified as below and the bn2, bn1 layers are removed. The **accuracy** achieved is **89%** and the **psum_recovered error** is **-2.97e-7**

layer1(0) conv1 - 8 input channel, 8 output channel

layer1(0) conv2 - 8 input channel, 8 output channel

layer2(0) conv1 - 8 input channel, 16 output channel



Generated activation.txt, weight.txt, psum.txt and output.txt for both VGG16 and RESENT20 from the squeezed layers. These .txt files served as the inputs to our testbench to test our design.

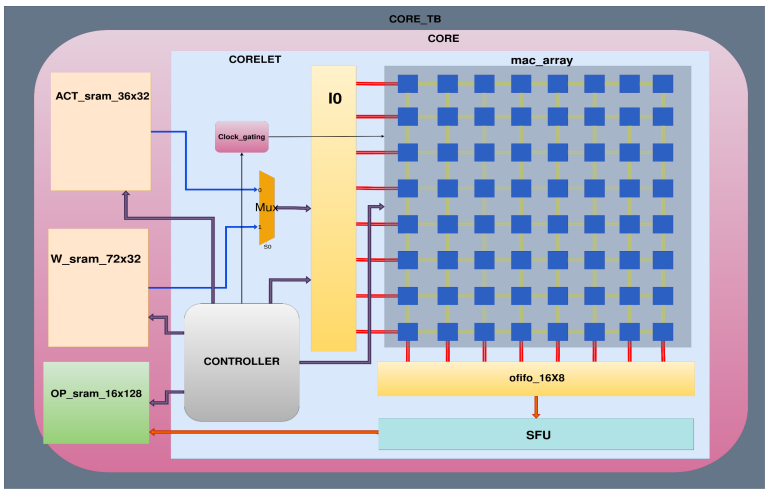
Table below shows the number of lines present in each file for VGG16 and RESNET20

	VGG16	**RESNET20**
Weight.txt	9 (3x3 kernel)	9 (3x3 kernel)
Activation.txt	36 (6x6 input channel dim)	1156 (34x34 input channel dim)
Residual.txt	NA	1024 (32x32 output channel dim)
Output.txt	16 (4x4 dim after ReLU)	1024 (32x32 dim after ReLU)

Inference: VGG16 achieves higher accuracy than RESNET20 when we tweak a layer to be 8x8. The reason is that in the case of VGG16, the tweaked layer is towards the end of the network, hence the accuracy drop is not significant, whereas in the case of RESNET, the tweaked layer is present in the start of the network, hence impacting the accuracy.

Part-2: Complete RTL core design

Figure **1** shows our systolic array architecture. It directly handles the 27th layer of VGG net (4x4 input channel dimension and 4x4 output channel dimension), as it computes on 8 input and 8 output channels with 36 activations per input channel (we show in the next section how we utilize it for the RESNET layer).



[FIG 1]

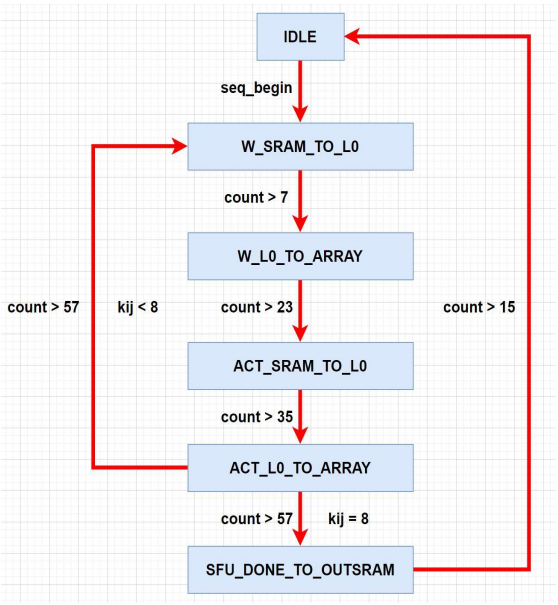
Hardware Structures

It contains three SRAMs – each for storing weights, activations, and outputs. Each row of weight_SRAM contains eight elements (each four bit) corresponding to eight output channel kernels. Eight such rows make up one K_{ij} of all the output channels. And nine such sets make up entire kernels. Thus, the size is 72x32. For the activation SRAM, since there are 36 activations per channel, 4-bit each, the size is 36x32. Finally, for storing 16 elements per output channel, each element with 16-bit precision, we chose the optimal SRAM size as 16x128.

The depth of each of the eight FIFOs in L0 is 64 and in the OFIFO is 16. Optimal sizes that we calculated are 36 and 8, but just to be safe we took the next power of 2.

Our SFU directly communicates with OFIFO, computes the convolution by adding respective partial sums for each element of the output channel matrix using a look-up table, applies ReLU, and writes out the final convolution to output SRAM. The SFU has 128 sfu_reg_banks (8 output channels, each having 16 elements) i.e sfu_reg_bank[0][15:0] for output channel 0, sfu_reg_bank[1][15:0] for output channel 1 and so on. Each sfu_reg_bank[x][y] accumulates the psum for one element of the output channel. The accumulator logic inside the SFU checks the LUT and directs the psum from OFIFO to be added to the correct sfu_reg_bak[x][y]. Detailed working of the SFU is explained in the alpha section.

Controller (FSM)



[Fig 2] FSM

Figure **2** shows the finite state machine implemented within the corelet.

- In the **IDLE** state, weights are loaded to the weight SRAM followed by a sanity check done by testbench (more on that in next section). The same is done for activations.
- Then, seq_begin is pulled up from the testbench and state changes to **W_SRAM_TO_L0**, where 64 weights are transferred to L0.
- Next, in state **W_L0_TO_ARRAY**, these weights are loaded into the mac array.
- **ACT_SRAM_TO_L0** and **ACT_L0_TO_ARRAY** do the same two operations for activations, with the instruction bits set appropriately.
- The **SFU_DONE_TO_OUTSRAM** state does the convolution by adding respective psum and applying ReLU. Finally, output from SFU is written to SRAM in 16 cycles, one for each O_{ij}.

Inference: We realized that moving the control logic from the testbench and implementing a controller to handle that makes our design as per industry norms. We also identified places to optimize the psum accumulation steps and hence saved on memory size and computation cycles.

Part-3: Test-bench Generation

(Compile with : **iverilog -g2012 -o compiled -c)**

From a functional perspective, we observed that the systolic array's operation does not require a lot of external control and thus we made it as standalone as necessary. Our testbench primarily controls data loading to the core, signals it the start of computation, and verifies the results. The rest of the intricacies are handled by the FSM as explained above. While this is sufficient for a network that maps perfectly to the array, like the layer of VGG16 in our case, data rearrangement and tiling is done in the testbench when the input cannot simply fit in our version of Systolic Array.

a. VGG16

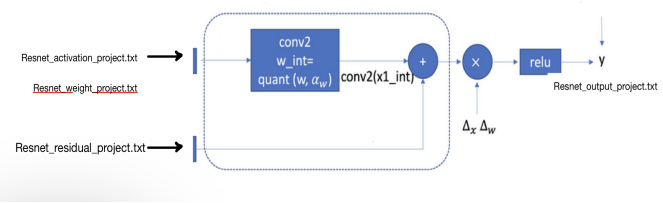
The testbench controls W_SRAM, ACT_SRAM OP_SRAM Inputs, CEN, WEN signals and Seq_start, dut_cl_sel signal which decides whether it will select addr, cen,wen from testbench or the FSM from corelet (2:1 mux). This allows us reading and writing controls for SRAMs from both ends. The core_tb receives Seq_done signal as output from core/u_corelet_inst.

The **core_tb** simulates core by controlling in a following manner :

1. Reading Weight txt file storing in a array (CEN =0, WEN = 0)
2. Writing to a w_sram and Taping each address to verify (w_sram Data match) (CEN=0,WEN=1)
3. Reading Activation txt file storing in a array (CEN =0, WEN = 0)
4. Writing to a act_sram and taping each address to verify (act_sram data match) (CEN=0,WEN=1)
5. Switched to control from FSM (corelet) side to read SRAMs (dut_cl_sel = 0)
6. Sequence Begin (fsm seq_begin)
7. Sequence Done (fsm seq_done)
8. Reading Output txt file storing in a array (CEN =1, WEN = 1)(dut_cl_sel = 1)
9. Final output Data compare (Output_data match)

b. ResNet

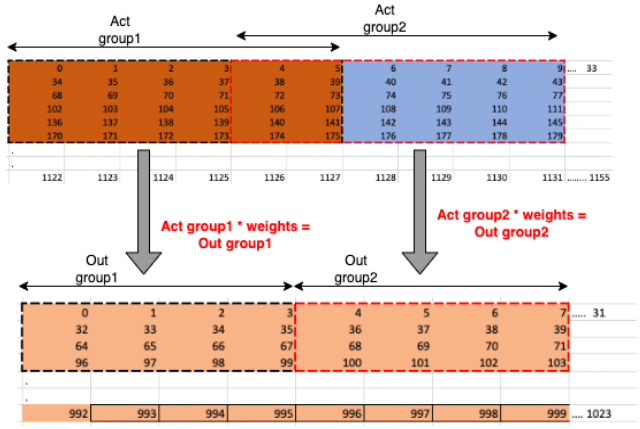
The testbench for Resnet has the same control signals as explained above for VGG16, with an additional logic to do tiling of the activations.txt and output.txt. Since our design can handle only 36 activations and 16 outputs at a time, we need to divide the inputs (1156 activations and 1024 outputs in the case of RESNET) into many sub-groups (64 for the case of RESNET) and send each sub-group through the design (**Iterating 64 times from Step 3 to 9** for VGG16).



[FIG 3]

Tiling code steps (python code attached for reference)

1. Arrange the input activations such that group1 is 0..5, 34..39 and so on as shown in the image. Create all such groups (each group has 36 elements, total 64 groups)
2. Arrange the output such that group1 is 0..3,32…35 and so on (total 16 elements)as shown in the image. Create all such groups (each group has 16 elements, total 64 groups)
3. Send each corresponding group of activations through the design, and compare the design output to the corresponding output group.
4. Figure 4 and 5 depicts the idea and python code.



[Fig 4]

```
hardware_ni_dim = 6
hor_step = hardware_ni_dim-kernel_dim+1 ## 4
ver_step = (hardware_ni_dim-kernel_dim+1)*a_pad_ni_dim ## 136
stop_point = (a_pad_ni_dim-hardware_ni_dim)*a_pad_ni_dim+1 ## 953
group_count = 0
act_arr = []
## Vertical movement loop
for v in range(0,stop_point,ver_step):
    ## Now move horizontally
    for h in range(v,v+34,hor_step):
        if h+hardware_ni_dim>v+a_pad_ni_dim:
            break
        group = []
        group_count+=1
        for hh in range(h,h+(hardware_ni_dim)*a_pad_ni_dim,a_pad_ni_dim):

            for hhh in range(hardware_ni_dim):
                group.append(hh+hhh)
        act_group.append(group)
```

[Fig 5]

Figure 6,7 shows snippets from verification results for VGG16 and RESNET. Complete results can be found in 1 . **DATA_MATCH_OP.log,** **2. core_tb.vcd,** **3. Compiled** files in respective folders.

```
100 25-th ACT read data is 00030400 --- Data matched
101 26-th ACT read data is 00050200 --- Data matched
102 27-th ACT read data is 00050000 --- Data matched
103 28-th ACT read data is 00040300 --- Data matched
104 29-th ACT read data is 00000000 --- Data matched
105 30-th ACT read data is 00000000 --- Data matched
106 31-th ACT read data is 00000000 --- Data matched
107 32-th ACT read data is 00000000 --- Data matched
108 33-th ACT read data is 00000000 --- Data matched
109 34-th ACT read data is 00000000 --- Data matched
110 35-th ACT read data is 00000000 --- Data matched
111
112  0-th Output data from OP_SRAM is 004c00e1000000680000000000000001f --- Data matched
113  1-th Output data from OP_SRAM is 00cc013600000045000000000000083 --- Data matched
114  2-th Output data from OP_SRAM is 00f601620000002b000000000000005c --- Data matched
115  3-th Output data from OP_SRAM is 00d700b40000000000000120000057 --- Data matched
116  4-th Output data from OP_SRAM is 000000ac0000000000000000000003 --- Data matched
117  5-th Output data from OP_SRAM is 00000133000000000000000000000180 --- Data matched
118  6-th Output data from OP_SRAM is 0000018600000000000000000000e7 --- Data matched
119  7-th Output data from OP_SRAM is 000000f60000000000000f00000057 --- Data matched
120  8-th Output data from OP_SRAM is 0000005300000000000000000003007c --- Data matched
121  9-th Output data from OP_SRAM is 00000090e000000170000000100015011b --- Data matched
122 10-th Output data from OP_SRAM is 000000c400000004000000000000095 --- Data matched
123 11-th Output data from OP_SRAM is 0000004a0000000000000000000003 --- Data matched
124 12-th Output data from OP_SRAM is 00000003d000000010000000000005d --- Data matched
125 13-th Output data from OP_SRAM is 000000640000000d000000000000ba --- Data matched
126 14-th Output data from OP_SRAM is 0000004700000000000000000000077 --- Data matched
127 15-th Output data from OP_SRAM is 00000000000000000000000000000022 --- Data matched
128
129 ** VVP Stop(0) **
130 ** Flushing output streams.
```

[FIG 6].VGG16 Data Match

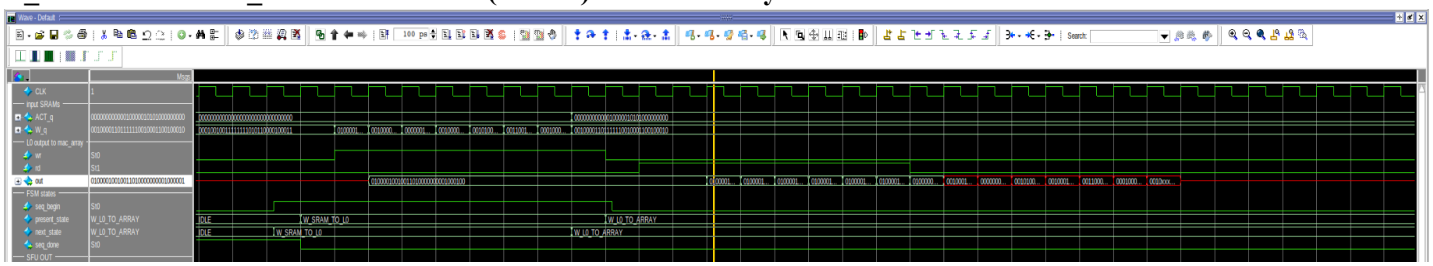
Nij = 6x6 (Input dim), Kij=3x3(Kernel dim), Oij = 4x4(Output dim)
8 Input channels ---> 8 Output channel

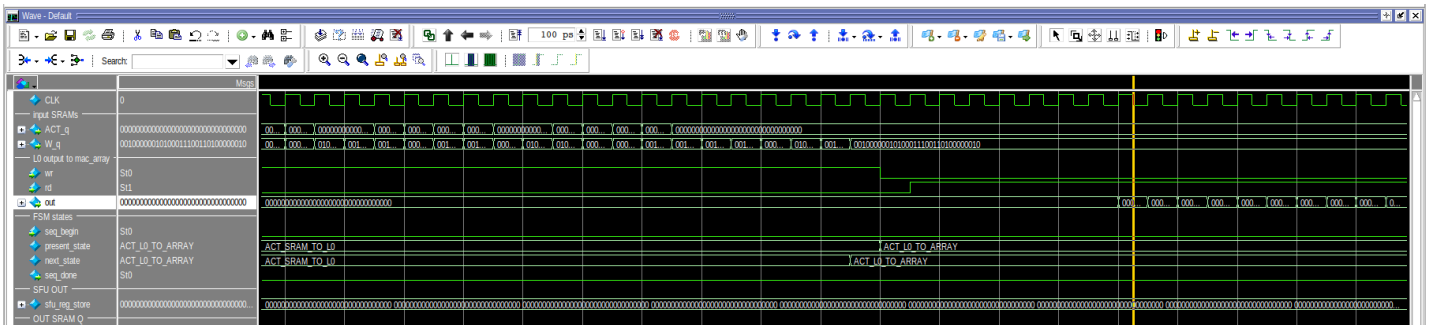
```
1180 **********  ITERATION   62 of 63 RESNET    *************
1181
1182  0-th read data from OP_SRAM is 00000027000000f6001e005c00000063 --- Data matched
1183  1-th read data from OP_SRAM is 0000000000000116001a009600000009a --- Data matched
1184  2-th read data from OP_SRAM is 000000000000009e005f005100000055 --- Data matched
1185  3-th read data from OP_SRAM is 0000009d000000fa0162000000a50000 --- Data matched
1186  4-th read data from OP_SRAM is 00000000000000f8000000bb000000b4 --- Data matched
1187  5-th read data from OP_SRAM is 00000000000000ef006b003b0000001b --- Data matched
1188  6-th read data from OP_SRAM is 000000140000000bc00b8000000000000 --- Data matched
1189  7-th read data from OP_SRAM is 001b009700790188000bf000000000000 --- Data matched
1190  8-th read data from OP_SRAM is 00000000000000a900bc00000000003c --- Data matched
1191  9-th read data from OP_SRAM is 0000006b000000e400e5000000000000 --- Data matched
1192 10-th read data from OP_SRAM is 000000e9000000bc0166000000b10000 --- Data matched
1193 11-th read data from OP_SRAM is 00000059000006200ec000000390000 --- Data matched
1194 12-th read data from OP_SRAM is 0000002a006100d80036002000000074 --- Data matched
1195 13-th read data from OP_SRAM is 003f006d008c00a2002900030000003d --- Data matched
1196 14-th read data from OP_SRAM is 005b010f006000dc008400000004f0005 --- Data matched
1197 15-th read data from OP_SRAM is 0000005a001700620024001700000000 --- Data matched
1198
1199 **********  ITERATION   63 of 63 RESNET    *************
1200
1201  0-th read data from OP_SRAM is 00140090007e011800380041000000000 --- Data matched
1202  1-th read data from OP_SRAM is 0000001100000f1004f002c00000065 --- Data matched
1203  2-th read data from OP_SRAM is 002a000e0090008b0000013e000000c8 --- Data matched
1204  3-th read data from OP_SRAM is 00000000000001d00ab000000330088 --- Data matched
1205  4-th read data from OP_SRAM is 0000001e000000540094000000000000 --- Data matched
1206  5-th read data from OP_SRAM is 0000004300000097000fa000000420000 --- Data matched
1207  6-th read data from OP_SRAM is 00930000000000000000016b00000088 --- Data matched
1208  7-th read data from OP_SRAM is 0000005200a0013d0079000000000148 --- Data matched
1209  8-th read data from OP_SRAM is 0000003a0000009100a0000000000000 --- Data matched
1210  9-th read data from OP_SRAM is 0007005100d4016a002400bd000000a6 --- Data matched
1211 10-th read data from OP_SRAM is 00000000000000000f400000006c0000 --- Data matched
1212 11-th read data from OP_SRAM is 006e0000000d3006f000001040000111 --- Data matched
1213 12-th read data from OP_SRAM is 0000000000210075000000900000067 --- Data matched
1214 13-th read data from OP_SRAM is 00050000000f0008000000c700000064 --- Data matched
1215 14-th read data from OP_SRAM is 00000029004601250 0b900000000065 --- Data matched
```

[FIG 7]. ResNet Data Match

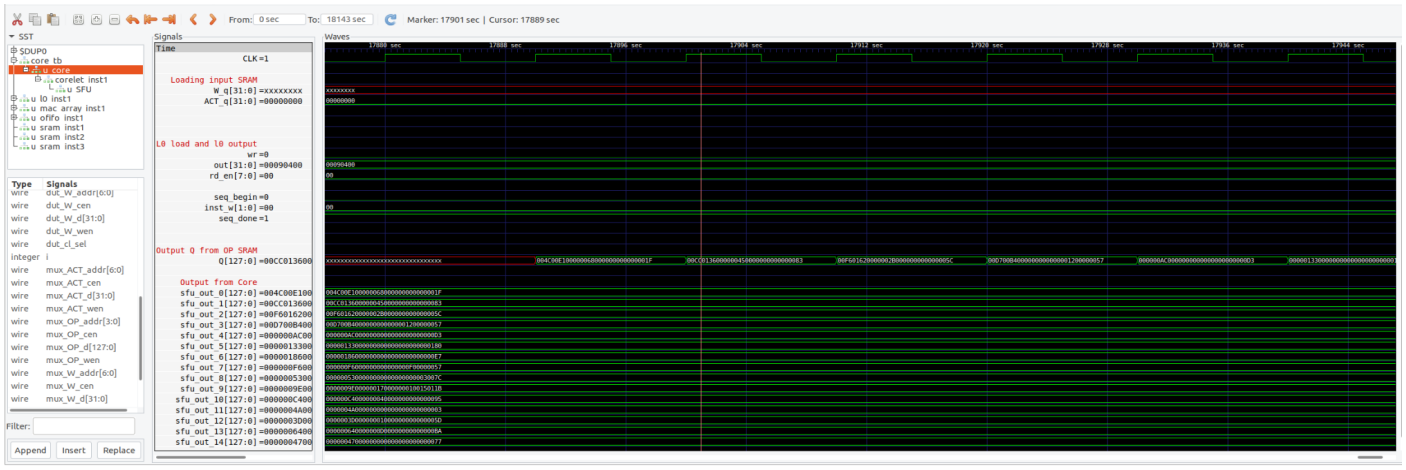
64 ITERATIONS (as there are 64 Tiles) Nij = 34X34, Oij = 32X32
Each output is 4x4 total computing 64 outputs;
8 Input Channel ---> 8 Output channel

Stimulus outputs: MODELSIM AND ICARUS (iverilog)
W_SRAM and ACT_SRAM to Corelet (l0 load) & L0 to Array load:





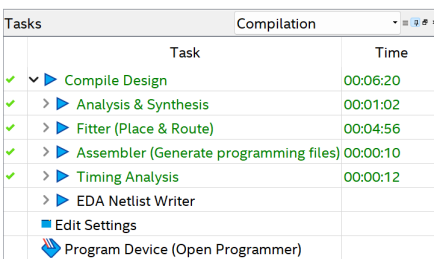
OP_SRAM Output Compare on Icarus (GtkWave)



Inference :
Data matched for VGG16 and ResNet (FIG 5,6). DONE signals can be added to each state, and tying it to the testbench, which will reduce the number of cycles between the OP_SRAM_q being available and the data match operation in the testbench.

Part-4: Mapping on FPGA (Cyclone IV GX EP4CGX150DF31I7AD)

We mapped the **corelet** and hierarchy underneath on the FPGA. Figure 8, 9, and 10 shows successful synthesis, fmax at the slow corner, and power consumption at 20% activity factor, respectively. The results are described in detail in the summary section.



Tasks	Compilation	
	Task	Time
✔	Compile Design	00:06:20
✔	Analysis & Synthesis	00:01:02
✔	Fitter (Place & Route)	00:04:56
✔	Assembler (Generate programming files)	00:00:10
✔	Timing Analysis	00:00:12
	EDA Netlist Writer	
	Edit Settings	
	Program Device (Open Programmer)	

[Fig 8]

Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Sun Nov 27 00:40:52 2022
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	corelet
Top-level Entity Name	corelet
Family	Cyclone IV GX
Device	EP4CGX150DF31I7AD
Power Models	Final
Total Thermal Power Dissipation	265.84 mW
Core Dynamic Thermal Power Dissipation	16.07 mW
Core Static Thermal Power Dissipation	119.37 mW
I/O Thermal Power Dissipation	130.40 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

[Fig 9]

Slow 1200mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	105.82 MHz	105.82 MHz	clk

[Fig 10]

Inference: In this part, we learned how to synthesize our design on an FPGA and analyze results like performance, power, and resource utilization. As per our observation, because the LUT size is 324 the time taken to access the LUT and writing to SFU reg bank is adding a lot of combinational delay, hence limiting fmax.

Part-5: +Alpha

a. FSM corelet (Controller):

Controlling Data traversal inside core and removing dependencies from test bench reduces IO from core_tb to core/u_corelet_inst thus increasing performance metrics as compared to normal implementation. (as explained above in Part-2 FSM).

b. Testbench modification to handle RESNET20 inputs:

As explained in Part-3 (Test-bench Generation), we wrote tiling logic inside the testbench to test all the **1024 outputs** (32x32 output dimension) for the RESNET layer. We have used that same hardware for VGG16 (which is for 4x4 input dimension and 4x4 output dimension) and have tested the RESNET20 inputs. Detailed logic and code has been given in Part-3.

c. Mac_Array Clock Gating :

We employ clock gating technique to gate the clock to the mac_array during all the FSM states other than W_L0_TO_ARRAY and ACT_L0_TO_ARRAY. This will conceptually reduce dynamic power of the design. We could not test the benefit of this idea since we do not have SAIF or FSDB files.

d. SFU - Filtered accumulation

There would be 36*9 (nij*kij) = 324 total partial sums out from OFIFO for one output channel, hence we used a LUT with size is 324. In one kij iteration there would be 36 partial sums for one output channel; out of which only 16 would be of interest to us. We use the LUT to identify the useful 16 psums and the corresponding sfu_reg_banks to fetch these psums, discarding the rest. For example, sfu_reg_bank[0][0] (stores the (0,0) element of output channel 0) would accumulate the following elements mapped from the LUT: 0, 37, 74, 114, 151, 188, 228, 265 & 302 and add these all to calculate the (0,0) element of output channel 0. This optimization **eliminates the extra memory accumulation** in the naive method and eliminates the extra step to store and fetch the psum to and from the PMEM, **but limits the fmax**. The size of the PMEM to store the final output also reduces from 324*8 to 16*8.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	31	33	34	35

36	37	38	39	40	41
42	43	44	45	46	47
48	49	50	51	52	53
54	55	56	57	58	59
60	61	62	63	64	65
66	67	68	69	70	71

72	73	74	75	76	77
78	79	80	81	82	83
84	85	86	87	88	89
90	91	92	93	94	95
96	97	98	99	100	101
102	103	104	105	106	107

108	109	110	111	112	113
114	115	116	117	118	119
120	121	122	123	124	125
126	127	128	129	130	131
132	133	134	135	136	137
138	139	140	141	142	143

144	145	146	147	148	149
150	151	152	153	154	155
156	157	158	159	160	161
162	163	164	165	166	167
168	169	170	171	172	173
174	175	176	177	178	179

180	181	182	183	184	185
186	187	188	189	190	191
192	193	194	195	196	197
198	199	200	201	202	203
204	205	206	207	208	209
210	211	212	213	214	215

Results and Summary

The tables below summarize the performance metrics of our design.

Metric	VGG16	RESNET
Accuracy	92%	89%
Psum Error	-2.9×10^{-8}	2.97×10^{-7}

Design	Fmax	Power (dynamic)	GOPS/s	MOPS/W	MOPS/mm² *	# Gates
Naive	121.3 MHz	28.4 mW	1.90	1.46	0.228	15,345
Alpha	**105.82 MHz**	**16.07 mW**	**2.80**	**2.58**	**0.019**	17,655

*Our synthesized design utilizes 12% of logic cell area. The total logic cell area is about 15% of the total silicon area, which is 121 mm^2 for Cyclone IV GX. This corresponds to 0.019 MOPS/mm^2.