Report

Part1. Train VGG16 with quantization-aware training

1. Squeezing and Removing Batch Normalisation: We updated the following line in the VGG_quant file to pick a layer in the middle and update its input and output channels to '8'.
   Original Command: 'VGG16_quant': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
   Updated Command: 'VGG16_quant': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 8, '8i', 512, 'M', 512, 512, 512, 'M']
   We made the above changes by selecting the layers corresponding to features [24] and [27]. After changing the layer in the above command, we also removed the batch normalization function from our selected layer by adding a statement in the _make_layer function for x=8i.

```
elif x == '8i':
    layers += [QuantConv2d(in_channels, 8, kernel_size=3, padding=1),
               nn.ReLU(inplace=True)]
    in_channels = 8
```

2. The hyperparameters of the model were then updated to achieve an accuracy of 91% after updating the VGG model.

```
    adjust_list = [20, 50]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1

lr = 1e-2
weight_decay = 1e-3
epochs = 80
```

3. We grabbed the inputs of the layer corresponding to features[27] that was outputs[8]. It was then used in the rest of our project.

```
x_bit = 4
x = save_output.outputs[8][0] # input of the 2nd conv layer
x_alpha = model.features[27].act_alpha
x_delta = x_alpha/(2**x_bit-1)
```

4. The difference between psum_recovered and psum_ref was calculated to see the impact of quantized sending input to the network.

```
difference = abs( output_ref[0] - output_recovered[0] )
print(difference.mean())  ## It should be small, e.g.,2.3 in my trainned model
```

```
tensor(0.1779, device='cuda:0', grad_fn=<MeanBackward0>)
```

5. Comparing the prehooked input of the next layer with psum_recovered.

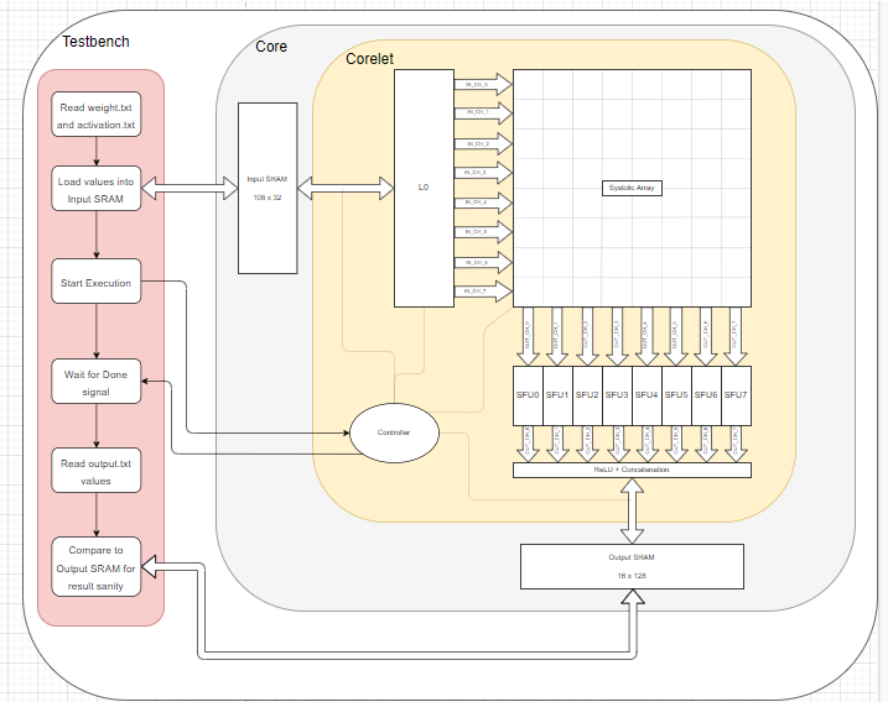
```
r=nn.ReLU()
next_layer_in_computed = r(output_recovered)
next_layer_in_ref = save_output.outputs[9][0]
difference = abs( next_layer_in_computed - next_layer_in_ref)
print(difference.mean())
```

```
tensor(1.3314e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

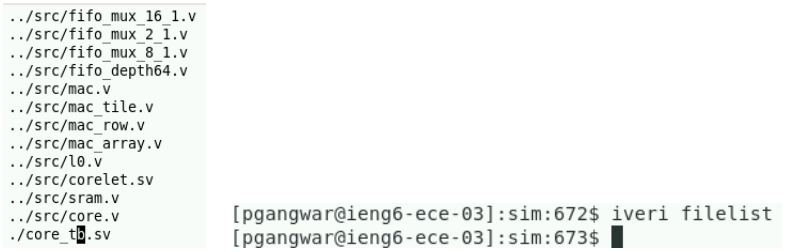
The first part of the project involved updating the VGG model to make sure it completely fits in the Systolic Array that we have designed. Retraining was performed to achieve an accuracy closer to the original one. After that the inputs of the chosen layer were grabbed, processed, and compared to the input of the next layer to check if we are indeed performing what we intended. The activations, weights, and psum of the layer were then dumped in a file to check the functionality of our hardware.

Part2. Complete RTL core design connecting following blocks

1. Some modifications on the vanilla version were performed by us because of which a block diagram of our proposal has been given below. We got rid of the output FIFO by using internal registers within the SFU to store the intermediate results (we are doing an in place accumulation of the psums generated by systolic array). It sends the outputs directly to the Output SRAM post completion of the execution. The integration of the blocks was done as per the block diagram.



2. The files inside the filelist and the compilation output is given below. SFU is implemented inside the Corelet.



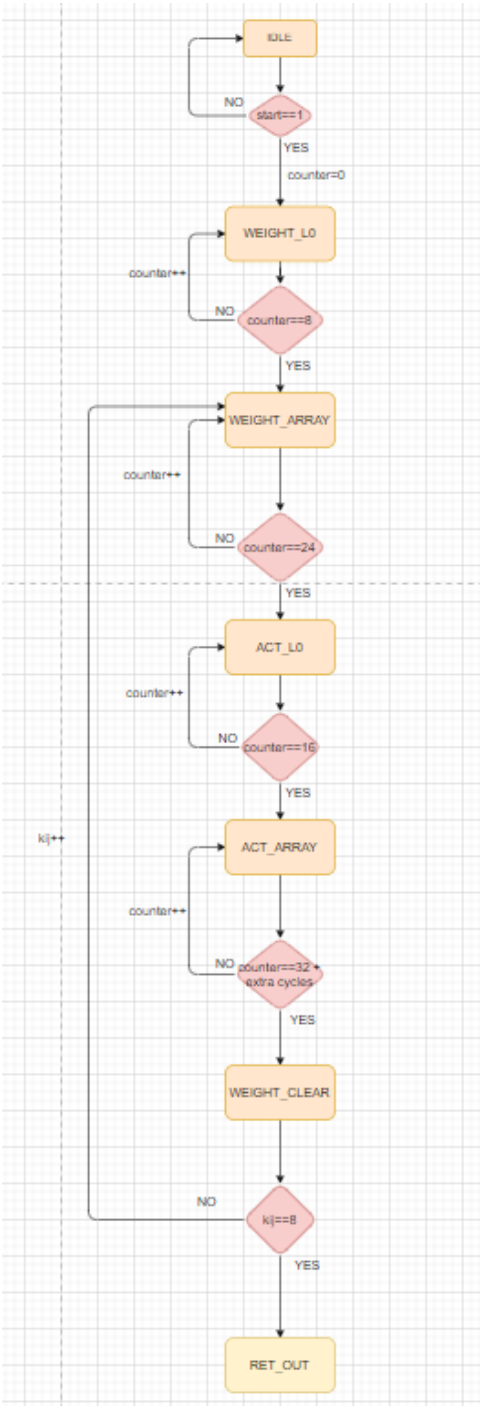
This part was about integrating the whole system by defining the interfaces and designing the controller.

Part3. Test bench generation to run following stages

We created a controller logic which is sitting inside corelet to handle the entire execution end to end.

Hence, we did not use the testbench provided by the professor. (we created a simple custom testbench for validating the outputs with txt files)

Controller:



1. L1 scratchpad loading for weight and activation (e.g., from DRAM, which is emulated by your testbench) --> state=WEIGHT_L0

   The memory interface to I-SRAM is activated with the memory address calculated by the controller logic. The addressing acquires the appropriate weight address from I-SRAM depending on the present value of kij. Eight weights corresponding to the output channels are read from SRAM and loaded into the L0.

2. Kernel data loading to PE register (via L0) : --> state=WEIGHT_ARRAY

The weights are transferred in a staggered fashion into the PE. Both the L0 and mac_array logic are designed to take care of the staggering internally. Controller gives both the “rd” signal to L0 and sets the “inst_w signal='b01” for 8 clock cycles.

3. L0 data loading --> state=ACT_L0

   Once the weight is transferred to the mac_array, we will read the appropriate activations from I-SRAM. The activations are placed after the weights (from address 72 onwards) in the SRAM. The controller selects the appropriate addressing mode between activation loading and weight loading based on the current state. Due to our special design, the controller sieves and acquires only 16 nij values corresponding to the current kij and loads them into the L0 block.

4. Execution with PEs --> state=ACT_ARRAY

   The controller sets the “inst_w='b10” for 16 cycles to the mac_array. At the same time, it asserts “rd='b1” for the L0. Due to this, the data being read from L0 will be transferred to the mac_array with proper alignment with the instructions. The staggering of data and instruction is taken care of by mac_array and L0 internally. After 32 clock cycles, the execution completes.

5. psum movement to L1 scratchpad (via OFIFO) -> N/A

   We are not storing the psums in the SRAM, nor are we using the OFIFO. We designed SFU with register-based memory to sum up the psums in place immediately after they are generated.

6. a) Accumulation in SFU and store back to psum SRAM --> state=ACT_ARRAY

   We have not created an OFIFO. Instead, we are storing the data in local registers present in the SFU. Since the data we acquire from mac_array is aligned, we can easily sum it up locally using an index pointer. After this, we move into WEIGHT_CLEAR state.

   b) Clearing the Weights stored in PE --> state=WEIGHT_CLEAR

   We pass a reset into the mac_array block that will make load_ready_q of each mac_tile to be 1. Essentially, we are making the PE ready to accept the next set of weights corresponding to the next kij. Then we go to ACT_L0 state to load new kij, if any left. Otherwise, we go into the next state to transfer the outputs to memory.

7. ReLU in SFU and store back to psum SRAM --> state=RET_OUT

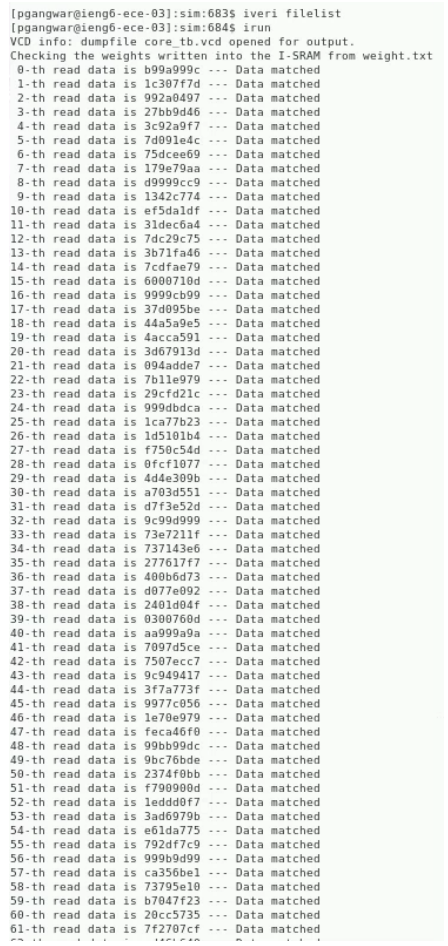
   If all the kij values from 0 to 8 have been covered then we know the data present in the SFU registers is the final post convolution output. Before sending data to the O-SRAM, we apply the ReLU function for each output channel using a comparator on the MSB. The addressing and CEN/WEN signals are taken care of by the controller. Since there are 16 outputs in our

case, the controller will only send 16 write cycles. We also pass a signal called done to let the testbench know that the execution is over, and the data has been stored in O-SRAM.

This part of the project was run with the next part that involves thorough verification of the whole system. We found the part of integrating everything together challenging and also a learning experience. We found a lot of bugs when we connected everything together in one shot so we went back and reconnected everything one at ab time which made things work.

Part4. Verification

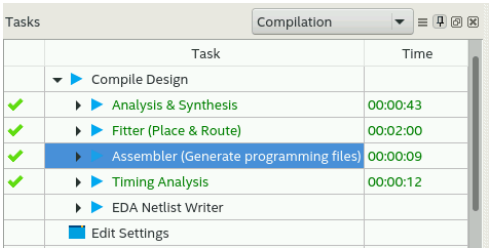
All the three files, “activation.txt”, “weight.txt”, and “output.txt” were generated during the first part of the project and read in the testbench. After running the testbench, we saw that the systolic array output matches the one from our python file.

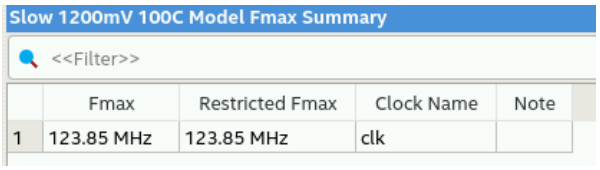


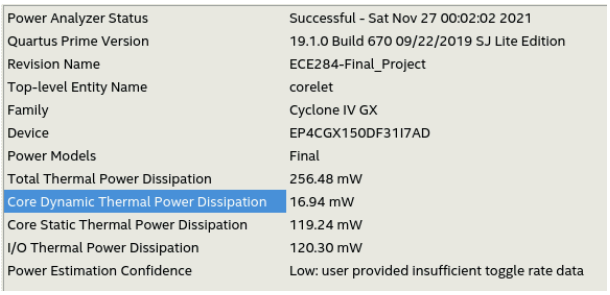
This part helped us verify the functionality of our Systolic Array by comparing it with the software’s output.

Part5. Mapping on FPGA

The output of various required tasks is shown below.







In this part of the project, we mapped the Corelet module that was designed to be synthesizable on the FPGA. The Corelet module did not have any memory instantiated within it and was just composed of logical blocks to allow it to be mapped to the FPGA. After running Place and Route on our design, we measured its power and maximum operating frequency in the slow corner. The key takeaway from this part was learning how to map the design on FPGA and measuring key design parameters, like frequency and power dissipiation.

Part6. Alpha

We implemented a few design optimizations over the baseline vanilla version.

1. A Mealy-FSM based controller inside the Corelet was designed that took care of
    a. the scheduling of the appropriate operations in L0 and systolic array
    b. memory address generation and interfacing with input and output RAM
    c. synchronization among different modules for the proper functioning of the design.

2. Designed a Sieving logic in the controller that only read the useful activation data from the I-SRAM. That means the only the psums that were actually utilized for calculating the final output corresponding to each kij we're computed. It helped by
    a. removing the unnecessary computations performed by the Systolic Array
    b. Since there are lesser partial sums, we saved the memory required for storing them
    c. reducing the energy consumption and latency as there are lesser memory and systolic array operations

3. Removing the unnecessary computation also simplified the addressing required for storing the partial sums and allowed for a parallelized structure for the output computation. Due to the index alignment in the psums provided after input Sieving, we were able to perform in-place accumulation of psums corresponding to different kij values in the SFU with just one copy of register array corresponding to the total number of outputs. Through this
    a. we saved the multiple write and read opertions to the SRAM
    b. Saved on the overhead of storing each psum separately in the memory for every kij. (Memory requirement became less by a factor of 9)
    c. Reduced latency in calculating final output. The finals outputs are generated within 1 clock cycle after systolic array completes execution.