Team Phoenix

Names	Achyuth Mahesh Esthuri	Devanshi Panchal	Harsh Rawat	Prashil Parekh
PID	A59026321	A59026424	A59023330	A59026482

Note: Please refer readme.txt, which states each file corresponds to which part.

Part 1) Training VGG16 with 4-bit quantization

- We modified the VGG_quant file to pick the 27th Layer and updated its in/out channels to be 8. Following this, we modified the hyperparameters to achieve an accuracy of 91.42%. The error between psum_recovered and prehooked input was 2.84 E-07.

```
prec = validate(testloader, model, criterion)
```

```
Test: [0/157]   Time 1.442 (1.442)      Loss 0.3054 (0.3054)    Prec 89.062% (89.062%)
Test: [100/157] Time 0.013 (0.027)      Loss 0.2183 (0.3487)    Prec 92.188% (90.996%)
 * Prec 91.420%
```

```
difference = abs( save_output.outputs[9][0] - psum_recovered )
print(difference.mean())
```

```
tensor(2.8440e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

- Here, we observed and inferred that the hyperparameters greatly influence the accuracy we can achieve with the model. After some experimentation, we reduced the batch size, increased the epochs and used learning rate decay technique to get our accuracy of 91.42%.

Part 2) Complete RTL design with connections

- We successfully designed the systolic array core with the following specifications for memory and fifos:
  - Activations and weights – Memory Size 2048*32 bits.
  - Psum memories – Memory size 4096*128 bits.
  - Storing final output – Memory Size 2048*128 bits.
  - L0 – Comprises of 8 FIFOS of 16 depth.
  - OFIFO - Comprises of 8 FIFOS of 16 depth.

Part 3) Testbench generation and verification

- Using the model we trained in Part 1, we generated the “activation.txt”, “weight_kij{}.txt”(For each kij) and “output.txt” files to drive as stimulus from our testbench to the DUT.
- We modified the testbench to achieve the following functionality:
  - We initially wrote all the activations from the “activation.txt” to L1 scratchpad memory by incrementing address(for activation) and driving the corresponding signals for write. Following this, for each kij, we wrote all the weights present in “weight_kij{}.txt” to L1 scratchpad memory starting at a different address.

- Next, the weights were written from L1 scratchpad memory to L0 by activating the corresponding signals. After this, the weights were transferred to the PEs in the systolic array by activating the load signal (Weight stationary mapping)
- Once weights were loaded, we transferred activation data from L1 scratchpad memory to L0. We optimized here **by taking fifo depth of 16**, due to which after a few clock cycles, we started driving the execute into mac_tile and ofifo_rd signals from the TB so that the while we are loading activations at the start, the execute can happen downstream (for psum) which can then be read by the OFIFO.
- After some cycles again, we start writing the psum data from OFIFO to the L1 memory. This process continues for all kij, where we store the computed psum for each starting at different address locations.(Eg: kij0 start address=100 in hex, kij2 start address=200 in hex ..)
- Once all psum data is in memory, we read the data from the different address locations (We generated an "acc.txt" file for the addresses from where to pick data to compute final out) and accumulated that data using sfp, applied relu and stored the final out back to memory.
- Finally, the L1 memory was read for the final out value and this was compared against the values present in "out.txt" file for completing the verification.

```
VCD info: dumpfile core_tb.vcd opened for output.
 0-th output featuremap Data matched! :D
 1-th output featuremap Data matched! :D
 2-th output featuremap Data matched! :D
 3-th output featuremap Data matched! :D
 4-th output featuremap Data matched! :D
 5-th output featuremap Data matched! :D
 6-th output featuremap Data matched! :D
 7-th output featuremap Data matched! :D
 8-th output featuremap Data matched! :D
 9-th output featuremap Data matched! :D
10-th output featuremap Data matched! :D
11-th output featuremap Data matched! :D
12-th output featuremap Data matched! :D
13-th output featuremap Data matched! :D
14-th output featuremap Data matched! :D
15-th output featuremap Data matched! :D
############ No error detected ##############
########### Project Completed !! ############
```

Part 4) Mapping on FPGA

- Here, we mapped the corelet module we designed on FPGA. Using Quartus Prime, we performed the steps of compilation, synthesis, place and route, assembler and timing analysis to measure the power (20% activity) and frequency at the slowest corner. We inferred here that the frequency can be improved by pipelining the design and the power consumed can also be reduced by optimization techniques which we have proposed in our alpha.

Specs	OPs	Freq(MHz)	DynPower(mW)	TOPs/GW	GOPs/s	Logic
VGG16	128	122.16	19.52	6.557	15.63	7545

Part 5) Multi-channel implementation in PE

- We initially changed the VGG16 model with the 27th layer to be 16x8 instead of 8x8. From this, we achieved an accuracy of 90.65% and used the activation, weight and out files to verify the RTL. Here, the activation and each weight file has 16 entries in each row, as opposed to 8 entries in each row for Part1-3 of the project.

```
prec = validate(testloader, model, criterion)
```

```
Test: [0/157]   Time 2.333 (2.333)      Loss 0.2969 (0.2969)    Prec 92.188% (92.188%)
Test: [100/157] Time 0.016 (0.045)      Loss 0.3865 (0.3302)    Prec 89.062% (90.656%)
 * Prec 90.650%
```

- The RTL was modified where the PE here stored weights of two input channels, two activations are loaded into a PE and MAC calculations for both the activations are calculated in the same PE.

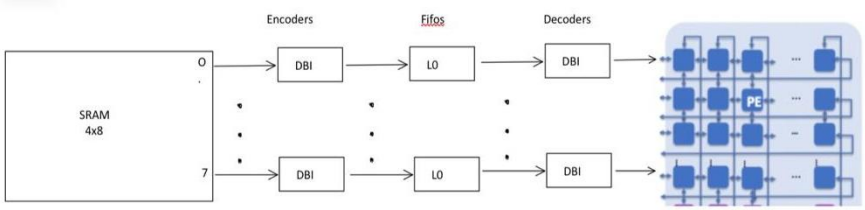
- The FIFO depth for the L0 was modified to 64 instead of 16 (Vanilla had depth of 16 for L0 FIFOs). The FIFO depth for OFIFO is still 16 only.
- We have also changed the logic generation for valid signals compared to the Vanilla version. Here, we are generating the valid signal for each PE individually, wherein, valid is high once a PE completes two MAC calculations and is ready to send the output data to the PE into the OFIFO.
- Changes were made to the TB as well so that we could drive the data properly to the core with multi-channel implementation. We added separate CEN signals for weights and activation in the activation SRAM. In the psum memory, we generated separate CEN and address signals for read and write.

```
VCD info: dumpfile core_tb.vcd opened for output.
 0-th output featuremap Data matched! :D
 1-th output featuremap Data matched! :D
 2-th output featuremap Data matched! :D
 3-th output featuremap Data matched! :D
 4-th output featuremap Data matched! :D
 5-th output featuremap Data matched! :D
 6-th output featuremap Data matched! :D
 7-th output featuremap Data matched! :D
 8-th output featuremap Data matched! :D
 9-th output featuremap Data matched! :D
10-th output featuremap Data matched! :D
11-th output featuremap Data matched! :D
12-th output featuremap Data matched! :D
13-th output featuremap Data matched! :D
14-th output featuremap Data matched! :D
15-th output featuremap Data matched! :D
############ No error detected ##############
########### Project Completed !! ############
```

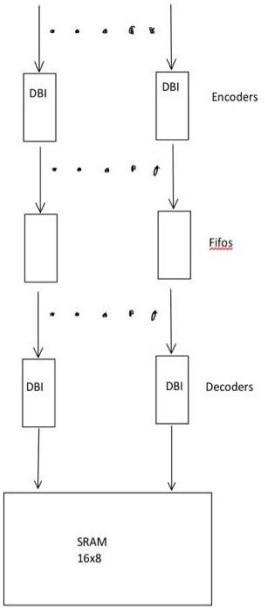
Part 5) Alpha

- The goal with our alpha was to reduce power consumed in the I/O and internal buses of the 2D systolic array due to constant switching of data as it flows through it. We do so by reducing the toggles to decrease the power consumption.

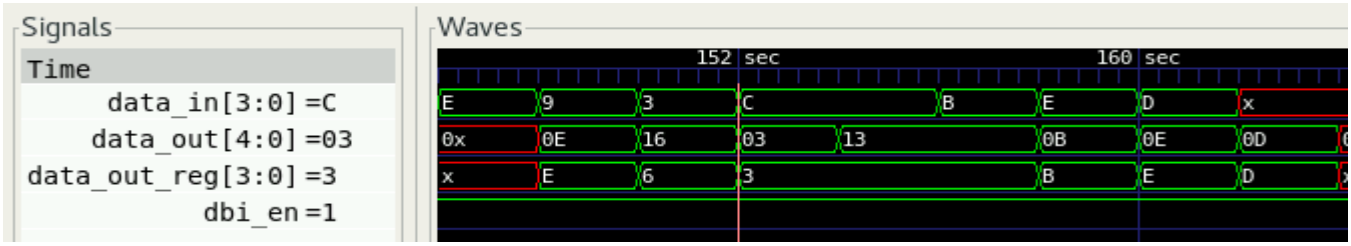
1. Data Bus Inversion (DBI)



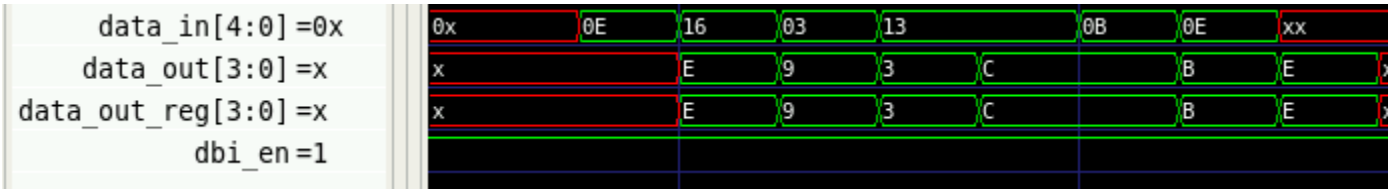
Using DBI to reduce the number of toggles between consecutive Data. We are using one decoder to decode the data such that the switching is less than 50 percent by XORing previous and current data and adding the bits to see the inversion. Before the SRAM PSUM memory, data is encoded and recovered.



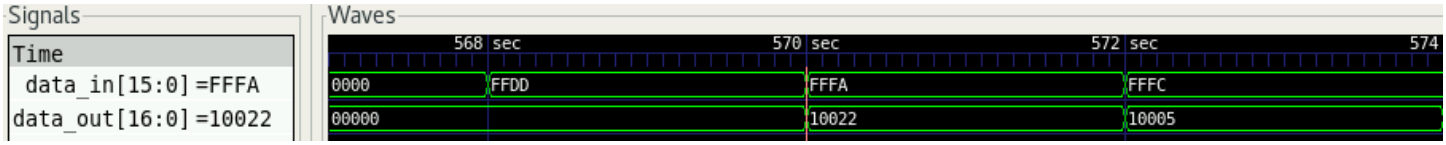
- **We verified the functionality of DBI with the full core and have provided a different TB for the same.**
- In the RTL, according to the diagram, we added the dbi encoder and decoder modules (Before and after L0 fifo and before and after OFIFO). Then, through the testbench, we drove the necessary data to ensure the DBI was performing correctly.
- As we can see , after the first data (E) The second data(9) has more than 50% toggles, and is encoded as 16 , the msb denoting it is the encoded data , so it gets decoded accordingly.



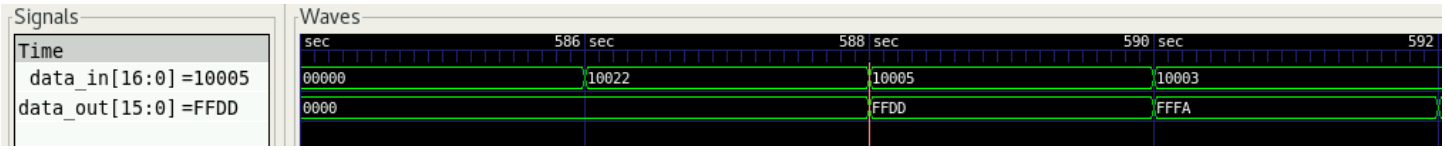
(DBI encode for weights and activation, coming from activation SRAM and going into L0 fifo)



(DBI decode for weights and activation, coming from L0 fifo and going into Mac array)



(DBI encode for psum coming from mac array and going into Ofifo)



(DBI decode for psum coming from ofifo and going into psum memory)

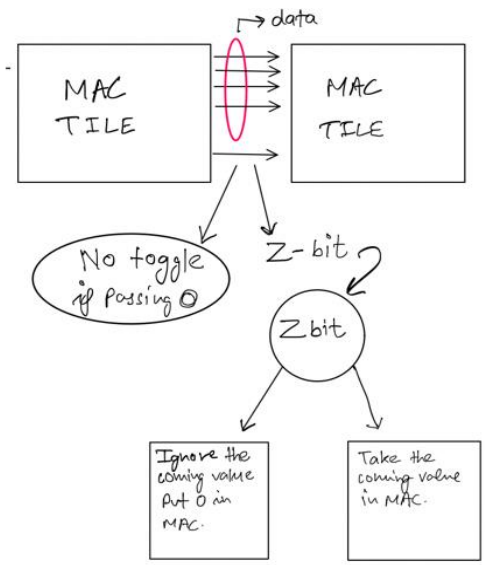
- We also mapped the RTL with DBI on Quartus Prime to measure the power (20% activity and 15% activity with DBI) and frequency at the slowest corner. As DBI reduces toggles to a max of 50%, we went ahead with a value of 15% for the toggles (between 10 – 20%). As we can see, with DBI, the power consumed is less than the vanilla version and the frequency of operation has increased as well due to the inherent pipelining.

Specs	OPs	Freq(MHz)	DynPower(mW)	TOPs/GW	GOPs/s	Logic
VGG16 with DBI(15% actv)	128	147.69	17.17	7.455	18.9	7804
VGG16 with DBI(20% actv)	128	147.69	22.88	5.594	18.9	7804

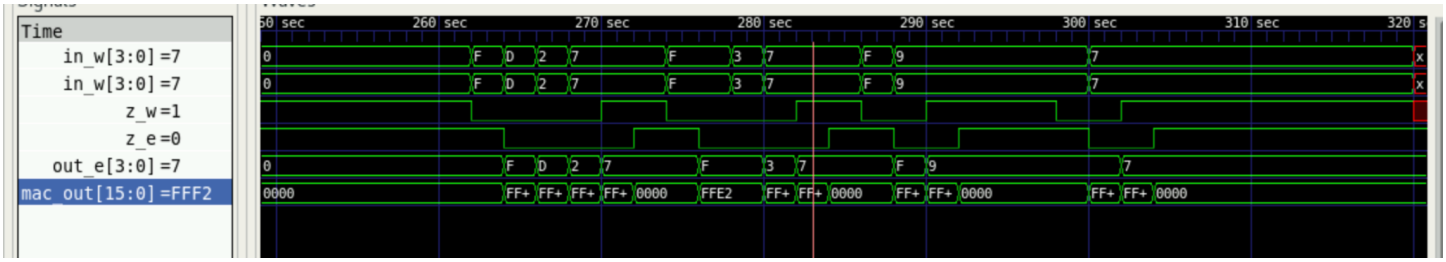
Pros	Cons
Power reduction(12%), Frequency increment (21%)	**Latency increase (5 clock cycles per core transition), area increase (3% logical elements increase)**

2. Toggle Reduction (TR) in MAC tile

- We reduce toggles in mac tile by not toggling activations or weights when either one of them is zero. We do so by looking at the z_bit. If the z_bit is 1, this means that the incoming activation/weight is zero, due to which the previous value is maintained and 0 is driven in the MAC tile.
- However, if the z_bit is zero, this means the incoming activation/weight is non-zero and hence should be used for mac calculations.



- In the RTL, according to the diagram, we modified the mac_tile to consider the z_bit and perform the functionality. Then, through the test bench, we drove the necessary data to ensure it was performing correctly. As we can see whenever the z_w is 1 , in_w coming from the previous mac_tile is not toggled, and zero is being given internally to the mac, so that when mac_out generated will be 0 when z_w is 1.
- Hence , we are saving power on the toggles whenever zero is getting passed without compromising on the functionality. (mac_out corresponds to out_s).



(Activations with value 0 going into mac tile)

- We also mapped the RTL with toggle reduction in Mac tile to measure the power (20% activity and 15% activity) and frequency at the slowest corner. As the sparsity of data increases, there will be more zeros and lot more toggles which can be reduced by our technique. We went ahead with a value of 15% for demonstrating the efficacy of our technique

Specs	OPs	Freq(MHz)	DynPower(mW)	TOPs/GW	GOPs/s	Logic
VGG16(TR, 15% actv)	128	121.95	15.28	8.377	15.6	7963
VGG16(TR, 20% actv)	128	121.95	20.37	6.284	15.6	7963

Pros	*Cons*
Power reduction when sparsity is high (21%)	**Area increase (5% increase in logic elements).**