ECE 284 Project Report

Part 1

In order to fit the design requirements of our 8x8 2d systolic array, we first had to make some adjustments to our quantization aware VGG 16 model. We changed the dimensions of the input and output channels of one of the convolutional layers to be 8x8 instead of 64x64. This was done to match the architecture of the systolic array, otherwise we would have to implement tiling. We also removed some of the consecutive layers, such as the residual layer in order to prevent dimension mismatch. After changing the dimension size and retraining the model, we were able to attain 92.67% accuracy with VGG16 on the CIFAR 10 dataset.

Part 2

To implement the RTL core design, we created a fully integrated design that connects key components, including the 2D systolic array with MAC units, scratchpad memories for activations, weights, and partial sums, as well as L0 and output FIFOs. A special function processor was also incorporated to handle operations like accumulation and ReLU. These components were wrapped in the corelet.v module, which encapsulates all submodules except the SRAMs. This hierarchical structure was essential for enabling seamless integration on the FPGA in later stages.

We designed the data flow to optimize memory access and reduce latency. The connections were verified for correctness, and the design successfully compiled without errors.

Part 3

To verify the functionality of our design, we developed a comprehensive testbench based on the provided core_tb.v template. The testbench simulated the behavior of the core by controlling its ports and emulating data transfer from DRAM. It executed a sequence of stages: loading weights and activations into input SRAM, transferring kernel data to the PE registers via L0, loading data into L0, performing computations with the PEs, moving partial sums to psum SRAM through the OFIFO, and finally processing the results through the special function processor for accumulation and ReLU.

We generated stimulus files (input.txt and weight.txt) and expected output files (output.txt) for the modified convolution layer with 8x8 channels. By running the testbench with these files, we validated the results against the expected output, ensuring zero verification errors.

Part 4

We synthesized our VGG 16 model in Quartus on the Cyclone IV architecture. We measured the frequency of our FPGA at the slow corner and found that it was operating at about 125.68 MHz. It was also using 27.4 mW of power when using a 20% input activity factor. From this, we can see that the model is able to perform at a very low power while achieving a strong performance. In terms of operations, it was able to perform 0.588 TOPs/W and 0.0161TOPs/s. The FPGA had 22,221 registers and 12,098 logical elements.

Part 5

We extended the functionality of our design by implementing a reconfigurable PE that supports both weight-stationary and output-stationary dataflows. The PE includes additional multiplexers to reroute data based on a control signal, allowing shared input, weight, and output registers to

operate in either mode. This required significant modifications to the array, core, and testbench to accommodate the dual dataflow functionality.

The testbench was updated to validate the reconfigurable PE with the first convolutional layer's inputs, weights, and activations. Since the array is limited to 8x8, only the first eight output channels and the first eight nij indices of the output feature map were verified. The results were compared to the estimated outputs from a PyTorch simulation, showing zero verification errors and confirming the correctness of the design.

Part 6

We also included various alphas to test our model in different ways. Our first alpha was refitting the Resnet20 model to be able to work with the 2d systolic array. With Resnet20, we were able to achieve 90.94% accuracy. Next, we implemented structured pruning to remove about 40% of the output channels. This meant that about ⅜ of the columns in our systolic array did not have to be computed, which would save us even more power. Finally, we implemented 8x8 tiling for the 64x64 convolutional layer in Resnet20. Since we are limited to an 8x8 array, we would have to compute 64 different tiles in order to match the 64x64 dimension of the convolutional layer.