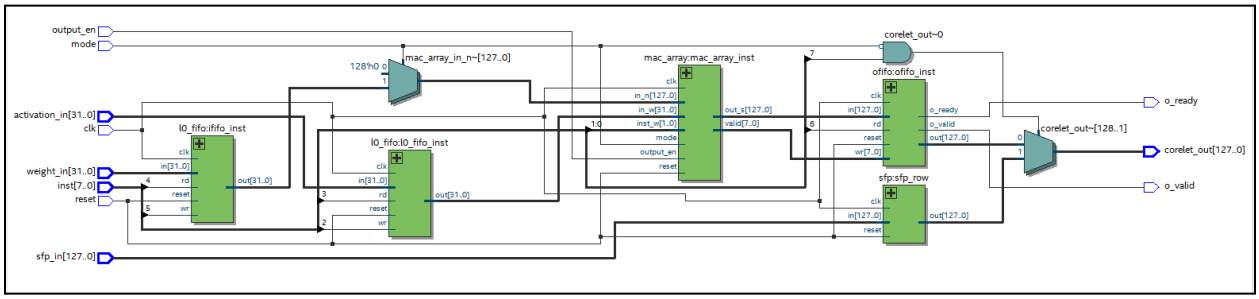
# ECE 284 Final Project: Reconfigurable AI Accelerator with 2D Systolic Array

Jacob Brown, Yuqi Lan, Kunal Bhandarkar

## Overview

This project aims to design a reconfigurable 2D systolic array based AI accelerator to achieve efficient, scalable, and performance AI inference on constrained hardware. Below is an image capturing our corelet:




In this report, we break down each step of our process.

## Part 1: Train VGG16 with Quantize-Aware Training

In this part, we trained a VGG16 model using quantization-aware training (QAT) on the CIFAR-10 dataset with 4-bit activations and weights, achieving over 90% accuracy. To optimize hardware mapping, we modified the convolution layer near features[27] by reducing its input and output channels to 8 and removing the batch normalization layer, simplifying the structure to “conv → ReLU.” This ensured efficient mapping onto an 8x8 systolic array without tiling.

We validated the design by computing psum_recovered after ReLU and comparing it with prehooked inputs for the next layer, achieving an error of less than $10^{-3}$. The results confirm that QAT preserves model performance under hardware constraints, demonstrating its effectiveness for integrating deep learning models with systolic array-based accelerators.

Our measures of success for this part were to achieve >90% accuracy and achieve an error rate <10^-3 for psum_recovered. In this part we achieved:

| Accuracy (CIFAR-10) | Quantization Error |
|---|---|
| 9019/10000 = 90.19% | 3.2451487e-07 |

## Part 2: Complete RTL Code Design

In this part of the project, we completed the RTL core design by connecting critical hardware blocks to form a functional processing unit. The design integrated a 2D array of MAC units with scratchpad memories for activations, weights, and partial sums (psum). Multiple banks were used for the psum SRAM to handle parallel data efficiently. The design also included L0 and output FIFOs, omitting the input FIFO (IFIFO) as weights were supplied west-to-east via the L0 module. Additionally, a special function processor was included for accumulation and ReLU operations.

The core design leveraged the modular structure of corelet.v, which encapsulates key blocks like L0, the 2D processing element (PE) array, and the output FIFO, excluding the SRAMs. This hierarchy ensured compatibility for FPGA implementation in Part 5 of the project. The success of this step was confirmed by completing all connections without any compilation errors, demonstrating the correctness and readiness of the RTL core design for further testing and mapping.

## Part 3: Testbench Generation

In this part of the project, the testbench was developed using the provided core_tb.v template to validate the functionality of the core.v module for a single 8x8-layer configuration. The testbench served as a controller, managing key operations and verifying the data flow through the design. The process began by loading weight and activation data into the input SRAM, simulating DRAM access. Kernel data was then transferred from the L0 module to the PE registers, followed by loading activation data through L0 into the processing elements (PEs). The PEs executed matrix operations, producing partial sums that were transferred to the psum SRAM via the output FIFO. These partial sums were further processed in the Special Function Unit (SFU) for accumulation and ReLU activation, with the final results stored back into the psum SRAM.

Stimulus files, including input.txt and weight.txt, and an expected output file, output.txt, were generated for the selected layer as part of the verification process. These files were applied to the testbench to simulate all operations, and the outputs were compared against the expected results. The testbench completed all stages with zero verification errors, confirming the correctness of the RTL design. Additionally, the design was verified to work seamlessly with TA-provided stimulus files, ensuring it meets the project requirements. This thorough verification process validated the functionality of the core design and its readiness for implementation in later stages.

## Part 4: Mapping on FPGA

In this part, we mapped the corelet.v design onto the Cyclone IV GX EP4CGX150DF31I7AD FPGA using Quartus Prime. The process involved synthesizing the design and completing placement and routing to ensure the core was correctly implemented on the FPGA. After successful mapping, we measured the operating frequency at the slow corner and calculated power consumption assuming a 20% input activity factor.

Our measure of success was reporting the final frequency, power, and specs in TOPs/W, TOPs/mm^2 and TOPS/s. All details of our original version mapping onto the Cyclone IV are in the following table:

| | |
|---|---|
| Fmax | 125.88 MHz |
| Switching Activity | 20% |
| PVT | 1200 mV 100C |
| Total Thermal Power | 229.58 mW |
| Total Logic Elements | 22,647 |
| Total Registers | 12,266 |
| Total Operations | 128 operations |
| GOPs/s | 16.11 G operations per second |
| TOPs/W | 557.53 operations per Watt |

# Part 5: Weight-Stationary / Output Stationary Reconfigurable PE

In this part, we designed a reconfigurable Processing Element (PE) capable of operating in both weight-stationary and output-stationary modes. The design required modifications to the PE array, core, corelet, MAC tile, MAC row, MAC array, testbench, and more to support these modes. Specifically, we incorporated an Input FIFO (IFIFO) to send weights into the PE array, unlike in Part 2. The PE design included multiplexers (muxes) to re-route data flow based on a 1-bit control signal, allowing the same input, weight, and output registers to be shared between the two modes.

For the verification, the testbench was designed to pass functional tests using the first convolution layer's input, weight, and activation data. Since the PE array was small, only the first 8 output channels and the first eight nij (coordinates) of the output feature map were mapped. The success of this part was determined by the zero verification error of RTL results compared to the estimated results from the PyTorch simulation. The design also passed testing with our input.txt, weight.txt and output.txt files and is compatible to be tested with a TA's own files of these type.

# Part 6: Alphas

## Alpha1: ResNet20

In reality, the convolutional layers are much bigger, so tiling and pruning are inevitable. To explore this we trained a ResNet20 model with 4-bit quantize-aware training, 80% unstructured pruning and fine-tuned it to 91% accuracy. We selected one of its 16x16 layers and split it into 2x2 tiles. As part 1, we validated that the quantization error was very small, the layer sparsity was about 81%. For comparability, the input is chosen as the 6x6 pixels at the top-left corner from the previous layer, making the sizes of input and output the same as part 1. Then we successfully recovered the output using a weight-stationary style in each tile. Finally we printed all the data in txt files for hardware tests. In Verilog, four tiles were processed by 2D systolic array serially and all psums were stored in sram. After execution, for each output, the SFP accumulated the 8 channels from psums of two vertical tiles at one time, and then combined two 8-channel outputs to a single 16-channel output. For sparsity-awareness, we just checked the values of activations and weights, and tried to skip the MAC operation when one of them is 0.

## Alpha2: FIFO Parallelism

For this alpha, we parallelized the FIFO loading and reading. In the weight stationary mode we load and read from the input FIFO during the weight loading of the systolic array thus saving around 7 cycles. We also parallelized the input FIFO during execution thus we reducing our cycle count by around the length of nij minus two. We also start reading the OFIFO once it is ready instead of waiting till after execution so we can save a considerable amount of cycles. For output stationary we also employ the FIFO load and read for the activation data to save us the length of nij minus two cycles. Our output stationary design includes a signal to start outputting the data and because of this we are able to output the data one row at a time. This allows us to output our data in only 8 (rows) clock cycles and means we do not need the OFIFO anymore. Thus, we clock gate the OFIFO when we are in output stationary mode for added power savings, bringing our power consumption closer to part 4.

| | Reconfigurable Array | Alpha 2 |
|---|---|---|
| Fmax | 115.55 MHz | 127.58 MHz |
| Switching Activity | 20% | 20% |
| PVT | 1200 mV 100C | 1200 mV 100C |
| Total Thermal Power | 234.83 mW | 230.25 mW |
| Total Logic Elements | 28,018 | 28,134 |
| Total Registers | 14,619 | 14,619 |
| Total Operations | 128 operations | 128 operations |
| GOPs/s | 14.79 G operations per second | 16.33 G operations per second |
| TOPs/W | 545.07 operations per Watt | 555.91 operations per Watt |

## Alpha3: Accumulation During OFIFO Read

In part 3 we accumulate each o_nij at a time after we collect all of the psums from the kernel loop. This results in an extra len_onij * len_kij cycles to accumulate our psums after execution. To eliminate these extra cycles we decided to accumulate right when we receive our psums from the OFIFO. We read the current partial o_nij from SRAM, add the current psum using SFP and save it back the next cycle. To do this we made a double buffered SRAM which can read and write in the same cycle. We calculate the reading address in our testbench and then the writing address is the past cycle's reading address (this is in theory but actually the delay is 2 cycles because the reading of psum needs 2 cycles). This saves us len_onij * len_kij cycles at the cost of slightly more expensive memory (double buffered). After execution, SFP reads o_nij again and performs the RELU function instead of accumulation.