Reconfigurable 2D Systolic Array-based AI Accelerator and Mapping on Cyclone IV GX
ECE 284 Group10

Zhongdongming Dai, Jheng-Ying Lin, Kejia Ruan, Yanghe Sun, Hongjie Wang
University of California, San Diego

1. Motivation

Our project optimizes CNN computation for hardware by leveraging quantization and reconfigurable PE. Aiming at RTL core design, FPGA mapping, and reconfigurable PE implementation, we went through efficient 8 × 8 array mapping, precise test bench designing, and completed verification and comparison with software results implemented with PyTorch.

2. Objective

2.1 Part1. VGG16 with quantization-aware training

• Train the VGG16 model using quantization-aware training with 4-bit input activations and weights, achieving an accuracy of over 90%.
• Optimize a convolutional layer (Feature27) by reducing its input and output channel numbers to 8 and removing the batch normalization layer that follows it.
• Map the optimized convolutional layer onto an 8 × 8 2D systolic array.

2.2 Part2. Complete RTL core design

• Design and integrate various components of the core, including a 2D array of MAC (Multiply-Accumulate) units, scratchpad memory, and a special function processor.

• Ensure the complete core design is free of compilation errors.

2.3 Part3. Test bench generation

• Develop a testbench to verify the implementation of all system stages.

• Generate stimulus and expected output files to validate the design, ensuring zero verification errors.

2.4 Part4. Mapping on FPGA

• Map the corelet.v design onto an FPGA using Quartus Prime.

• Complete the synthesis, placement, and routing processes while measuring the system's operating frequency and power consumption.

• Provide a final report detailing frequency, power metrics, and overall efficiency.

2.5 Part5. Weight-stationary and output stationary reconfigurable PE

• Implement a reconfigurable execution mode in each Processing Element (PE) to support both weight-stationary and output-stationary operation

• Ensure that the RTL verification results match the PyTorch simulation estimates with zero error

2.6 Part6. +alpha

• Add custom enhancements or techniques to improve the system.

• Conduct thorough verification of the enhancements and present the results.

3. Implementation

3.1 Part1. VGG16 with quantization-aware training

We trained VGG16 with quantization-aware training of 4-bit precision of input activation and weight on CIFAR10 dataset. And we got 92% accuracy on the test data.

3.2 Part2. Complete RTL core design

We successfully connected the 2D PE array containing MAC units to the core, ensuring seamless data flow. For the scratchpad memories, we implemented input SRAMs for activations and weights with proper connections and designed multiple p_sum SRAM banks for efficient

intermediate result storage. We integrated the L0 module to manage data flow from west to east and implemented output FIFOs to store results from the 2D PE array, omitting the IFIFO in this phase. Additionally, we designed and integrated a special function processor to handle accumulation and ReLU operations. Finally, we developed corelet.v, which includes all components except the SRAMs, ensuring compatibility with FPGA deployment. The modular design simplifies future FPGA mapping while keeping SRAMs external.

The complete RTL core design compiled successfully without errors. And all modules were correctly integrated and verified, ensuring functionality.

3.3 Part3. Test bench generation

We designed a testbench to verify the functionality of weight stationary mapping. The testbench reads data from activation.txt and weight.txt, passes it to core.v along with instructions, and compares the output with output.txt. If all values match, it confirms the correctness of the weight stationary mapping implemented in core.v.

The files activation.txt, weight.txt, and output.txt were generated with a Jupyter Notebook, which extracts the input, weights, and output from the 27th layer of our trained quantized VGG16 model, formatting them to meet the input requirements of weight stationary mapping. Additionally, during the accumulation phase in the SFU, we generated an address.txt file to identify which addresses should be summed.

The 34-bit instruction input for core.v, stored in inst_q, includes commands such as load, execute, read/write enable for L0, IFOFO, and OFIFO, as well as chip enable, write enable, and address settings for x_mem and p_mem. It also controls the acc and Relu operations in the SFU.

3.4 Part4. Mapping on FPGA

We used Quartus Prime to simulate mapping onto FPGA. The results of Quartus Prime simulation are given in Table I:

Cyclone IV GX FPGA Mapping of Vanilla Version	
FMax(MHz)	127.99
OP	128
GOPs/s	16.38
TOPs/W	3.76e-09
Core Dynamic Power(mW)	34.01
Logic Elements	22,414
Registers	12,098

Table I. FPGA Report

3.5 Part5. Weight-stationary and output stationary reconfigurable PE

Building upon Parts 2 and 3, we revised our design to incorporate a reconfigurable processing element (PE) architecture capable of supporting both weight-stationary and output-stationary dataflows. This update required corresponding modifications to all relevant modules, including *mac_tile.v*, *corelet.v*, *core.v*, and the testbench.

In the output-stationary mode, each PE no longer requires a separate data-loading instruction, as data loading and execution occur simultaneously. Additionally, partial sums are now held within a register inside each PE. To facilitate weight loading, we introduced an additional input FIFO and an associated SRAM module. Moreover, our testbench and input text files were updated to properly accommodate these changes.

3.6 Part6. +alpha

Several alphas are implemented in our project:

3.6.1 Alpha1: Unstructured and Structured Pruning

We used both unstructured and structured pruning techniques with 80% sparsity on VGG16 model to save energy and computation by increasing sparsity and reducing data movement.

And after pruning, the accuracy dropped to 10% in both cases, then we fine-tuned the two models using the same hyperparameters except for epochs. And we found that the unstructured pruned model can achieve 13% higher accuracy using almost 50% less epochs than the structured pruned model.

3.6.2 Alpha2 - Resnet20 Mapping with Quantization Aware Training

Our second alpha is to map the modified layer in the Resnet20 model with Quantization Aware training, which achieves 88% accuracy with 0.7054 quantization error.

We modified the whole second layer Sequential and extracted the second Basicblock ’1 st convolutional layer, whose parameters are shown in Figure 1.

```
layer = list(model.children())[4][1].conv1 # 2nd BasicBlock's 1st Conv2d layer
layer
```

```
QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
```

Figure1. Resnet Modified Layer

The input and output channels are the same (8) for the convenience of mapping on our 2D systolic array hardware. But some mapping difficulties we met were mainly caused by the verification process. Because this convolutional layer’s input feature map size is way bigger than the layer we mapped in Vgg16, (18*18 after padding) and output feature map is 16*16. Therefore, the partial sum address .txt needs 256 times 9 lines and each address needs 12 bits to store instead of 11 bits, so we need to modify many parameters in the test bench file including some components like SRAM.

3.6.3 Alpha3 - Concurrent OFIFO Read/Write

We implemented a strategy to utilize a concurrently read/write OFIFO buffer, which significantly reduces the required FIFO depth. This optimization allowed us to decrease the necessary depth from 64 to just 18, resulting in a substantial reduction in both hardware complexity and associated costs.

We modified the control flow to ensure that the testbench initiates reading from the OFIFO as soon as it detects the ofifo_valid signal, effectively constraining the FIFO depth to a smaller value.

By minimizing the FIFO depth, we not only streamlined the overall system architecture but also enhanced the efficiency of data handling within our processing pipeline. This adjustment is expected to lead to improved performance while keeping resource utilization at optimal levels.

3.6.4 Alpha4 - Random Input for Verification

To make the functional verification more robust, we supplemented the model's single-layer output with randomly generated activation and weight data. We computed the corresponding output with Python for comparison and validation. The successful verification with randomly generated data demonstrates that our code is not hard-coded for specific results but genuinely implements the functionality of the systolic array.

4. References

The detailed project description and resources can be found in the repository: VVIPLab/ece284fa24 on GitHub.