ECE284 Final Project

Sin-Yu Chen, Yu-Hsiang Tseng, and Ya-Chi Liao (Group Shin-Yu, Ya-Chi, Yu-Hsiang)

December 16, 2023

Basic Design

For Part 1 - training of VGG16 with quantization-aware training, we applied the 8 channel on Layer 27 and we achieved 91% accuracy with error of 5.2647e-07, all of which satisfy the requirements.The screenshot of the result are listed as follows.

```
,
(25): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(26): ReLU(inplace=True)
(27): QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
```

Figure 1: Location of the 8-Channel Layer

```
PATH = "result/VGG16_quant1130_2_newmodel/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda")

model.cuda()
model.eval()

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in testloader:
        data, target = data.to(device), target.to(device) # loading to GPU
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(testloader.dataset)

print('\nTest set: Accuracy: {}/{} ({:.0f}%)\n'.format(
        correct, len(testloader.dataset),
        100. * correct / len(testloader.dataset)))
```

```
Test set: Accuracy: 9127/10000 (91%)
```

Figure 2: Test Accuracy of VGG16 with quantization-aware training

```
difference = abs(out_ref - out_recovered)
print(difference.mean())
```

```
tensor(5.2647e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

Figure 3: Error of the Psum Recovered

For Part2-Part5, we finished the hardware design with the alphas we designed and the respective performances are listed in the following table.

	Basic Design	+Alpha 3	+Alpha 4
Frequency(MHz)	132.36	121.67	82.57
# of Logic Elements	17,538	19,486	17,210
Dynamic Power (mW)	46.99	42.47	27.13
TOPS/W (ops/(s·w)	0.181	0.183	0.195
TOPS (10^{-3}ops/s)	8.47	7.79	5.28

Figure 4: Performance of Mapping on FPGA

Alpha 1 - Split SRAMs

Implementation

In the original design, when writing weight and activation to memory, contention for memory access happens as they share the same SRAM. However, these two data are not inherently dependent on each other. Therefore, we resolved this false dependency by splitting the SRAM to enable both data to have individual access to memory and thus parallel write and read. The new block diagram is show in Figure 1.

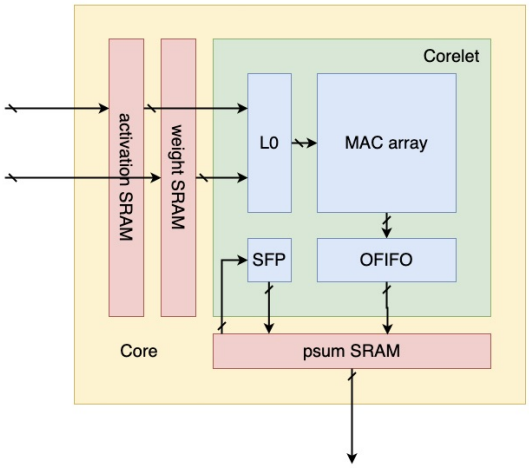

Figure 5: Block diagram for design with split SRAMs

Alpha 2 - Pipeline Scheduling

Implementation

To fully speed up the processing speed, we rescheduled the pipeline by altering the control signals in the core_tb.v which serves as a wrapper of the design. To elaborate, we introduced three improvements to the original pipeline. First, we start reading the weight values one cycle after the weight was written to L0 FIFO from SRAM. Second, We start executing the MAC array one cycle after the activation is written to L0. Third, we start writing the psum value to SRAM once OFIFO has collected all values in a single row. Together with the split SRAMs design, these improvements yield an estimated 85% speedup compared to the vanilla version. The scheduling diagram and the speedup table can be found in Figure 2 and Table 1, accordingly.

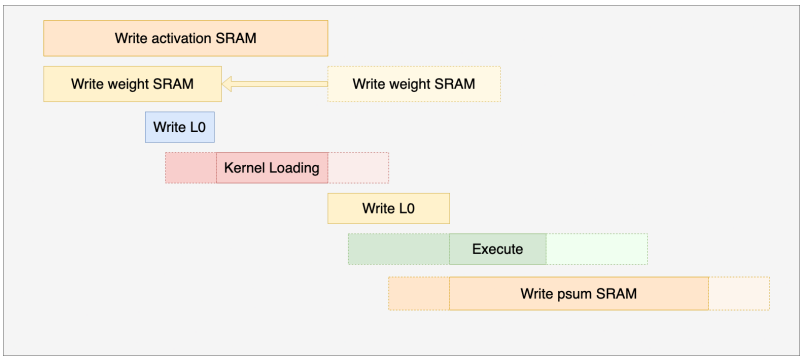

Figure 7: Scheduling diagram

Method	Cycle Count	Speedup
Vanilla	4*len_nij+ 5*col+2*row +O(1)	
Split SRAMs	3*len_nij+ 5*col+2*row +O(1)	~22%
weight pipelining	3*len_nij+ 4*col+2*row +O(1)	~28%
activation pipelining	2*len_nij+ 4*col+2*row +O(1)	~67%
psum SRAM pipelining	len_nij+ 6*col+3*row +O(1)	~85%

Table 1: Speedup table

Alpha 3 - Gating on MACs

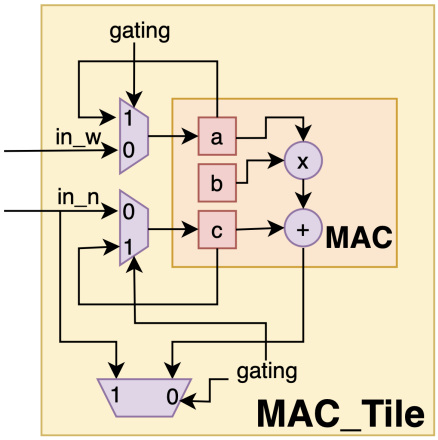

Figure 6: Scheduling diagram

We implemented gating in our MAC unit when in_w is equal to 0 to conserve power by excluding irrelevant operations. Unexpectedly, the overall power consumption increased. This unexpected outcome is likely attributed to our use of registers to store the current values of a and c. The increase in the number of registers also amplified the power consumed by registers, surpassing the power savings achieved by excluding certain operations.

Alpha 4 - Compression on FIFO

Motivation

The compression method introduced in class, such as RLC and Huffman Coding, require sequential compression or decompression. Besides, the resulting compressed lengths are not fixed, which make it hard to allocate memory. Our attempt involved devising a novel compression method that enables parallel compression/decompression and ensures a fixed compressed length. Consider an input to OFIFO, comprising 8 16-bit numbers and a total of 128 bits, with common patterns 0000 and FFFF.

We detected if there are 4 or more instances of these patterns within the 8 numbers. If so, we compress the rightmost 4 common patterns into 1 bit (0000 to 0 and FFFF to 1). An 5-bit index is included to indicate the permutation of compressed and uncompressed numbers. This compression results in 73 bits ($16 \times 4 + 4 + 5$ bits) for the 128-bit data. For cases where compression is not feasible (i.e., fewer than 4 common patterns), we utilize an auxiliary FIFO to store the entire 128 bits. Figure 10 illustrates our modified OFIFO and provides an example of the compression process. In practical implementation, the depth of the auxiliary FIFO is configured to be 1/2 the depth of the main FIFO, a setup proven effective with our test data.

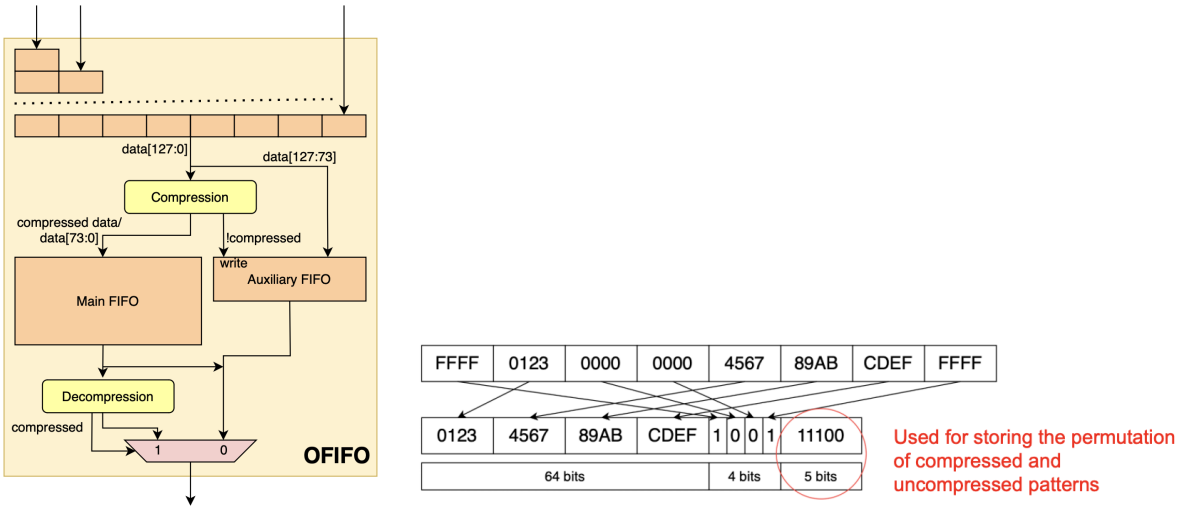

Figure 8: Proposed OFIFO and an example of compression

Result

Theoretically, we reduced the FIFO size by $(128 \times 64) - (16 \times 36 + 73 \times 64 + 55 \times 32) = 1184$ bits (14%), which is quite close to the FPGA mapping result that the total number of registers is 1197 less than the basic design (Basic: 12482, Compressed: 11285). Initially, we applied this compression method to OFIFO for functional testing, but it can also be extended to L0 or SRAM. Our approach also offers flexibility; for example, adjusting the threshold number of compressed patterns to 3 can compress more numbers but leading to a higher compression ratio ($16 \times 5 + 3 + 5 = 88$ bits). However, the effectiveness of our method confined to scenarios with some common patterns and a resulting long critical path needs further improvement through pipelining.

Alpha 5 - Pruning and 2-bit Quantization

Implementation

As for the training VGG16, we implemented both L1 structured and unstructured pruning introduced in class. The pruning rate were both 0.8. After pruning, we recovered the accuracy by training an extra 50 epochs and reached the recovered accuracy of 77.61% and 89.59% respectively.

	Original Accuracy	Sparsity	Pruned Accuracy	Recovered Accuracy	Sparsity after Quantization
L1 Structured	91.27%	0.8008	10%	77.61%	0.8313
Unstructured	91.27%	0.8	10%	89.59%	0.8133

Figure 9: Results After Retraining VGG16 for Structured and Unstructured Pruning

We also performed 2-bit quantization on all layers of VGG16 and achieved 83.36% of accuracy with quantization error of 2.7509e-07.

	Accuracy	Quantization Error
4-bit Quantization	91.27%	5.2647e-07
2-bit Quantization	83.36%	2.7509e-07

Figure 10: Training Result for 2-bit Quantization