CNNs on Multi-core 2-D Systolic Array

with Pruning and Huffman Coding

284 little group

Zhen Bian, Hanyi Chen, Mingyu Liu, Cheng Qian and Xin Zhao

Part 1: Train VGG16 with Quantization-aware Training

The training results are shown in the **Table 1**.

Table 1. VGG16 training results

	Requirement	Our model
Accuracy	> 90%	92.140%
PSUM recovered error	< 0.001	0.0006

Part 2-3: Completion of RTL Core Design and Core Stages

Workflow in Mac_Array

The Mac_Array employs the Weight Stationary approach for data processing. The process is as follows.

First, storing Input Activation: Input activations are stored in the lower bits of the XMEM module; second, loading Weights into Mac_Array: The Mac_Array mode is set to "load," enabling weights to be stored into the array; third, executing Convolution: After weights are loaded, the mode is switched to "execute," and the input activations are fed into the Mac_Array, where convolution is performed between the activations and the stored weights, producing the output; fourth, data Alignment in OFIFO: Due to the pipelined nature of the architecture, the output data from the Mac_Array is staggered. To address this, the OFIFO module ensures proper alignment.

Each column in the Mac_Array generates a valid signal upon outputting data. The valid signal activates the corresponding column in the OFIFO, allowing the data to be stored in the respective column of the OFIFO. The mechanism ensures the aligned transfer of data from the Mac_Array to the next stage.

Accumulation in SFP Module

The SFP module is responsible for retrieving and accumulating specific data from PMEM. The address of the data in PMEM is calculated using the formula below:

$$A_{\text{pmem}} = \left(\frac{i}{o_{\text{ni\_dim}}}\right) \cdot a_{\text{pad\_ni\_dim}} + \left(i \bmod o_{\text{ni\_dim}}\right) + \left(\frac{j}{k_{\text{i\_dim}}}\right) \cdot a_{\text{pad\_ni\_dim}} + \left(\frac{j}{k_{\text{i\_dim}}}\right)$$

$$\cdot a_{\text{pad\_ni\_dim}} + \left(\frac{j}{k_{\text{i\_dim}}}\right) \cdot a_{\text{pad\_ni\_dim}} + \left(j \bmod k_{\text{i\_dim}}\right) + \left(j \cdot \text{len\_nij}\right)$$

Where i is input index, $o_{\text{ni\_dim}}$ is output dimension for the ni axis, $a_{\text{pad\_ni\_dim}}$ is padded dimension for ni axis in activations, j is Kernel index, $k_{\text{i\_dim}}$ is kernel dimension for the ii axis, and len_nij is length of the nij dimension.

This accumulation step ensures the final output data is correctly computed and prepared for further use. And the output results are shown in the **Figure 1**.

Figure 1. Output results

Part 4: Mapping on FPGA

The results are shown in the **Table 2**.

Table 2. Characterizations of mapping

Characterization	Value
OPs	128
Frequency	132.82MHz
Dynamic power	35.72mW
GOPs/s	17
GOPs/w	0.00358
Logic elements	22126

Part 5: Weight-stationary and Output-stationary Reconfigurable PE

The Weight Stationary architecture and Output Stationary architecture is illustrated in the **Figure 2**.

Data is input from the Huffman Decoder, where it is first decoded and then pruned before being stored in the memory module XMEM. Subsequently, the data is transferred to the Mac_Array core via the FIFO L0. The output from the Mac_Array is aligned in the OFIFO and stored in the memory module PMEM. The SFP module then retrieves data from PMEM to perform accumulation. For the input data format, please read README.txt in Part5 folder.

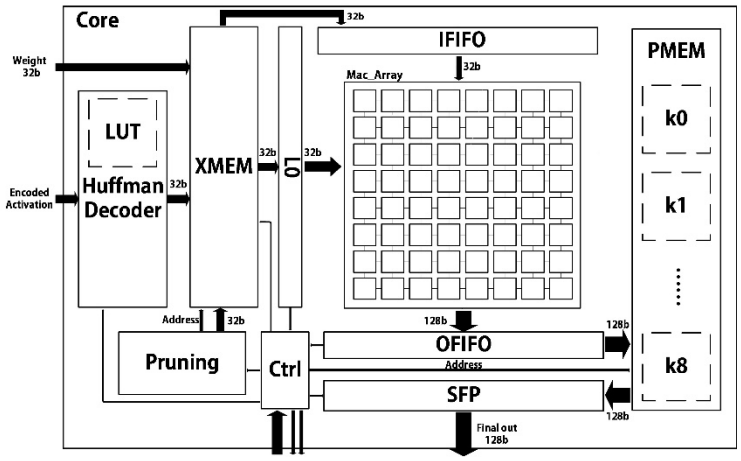

Figure 2. Configurable PE structure

Activation and weight data are first stored in the memory module XMEM. Then the activations are transferred to L0, weights are transferred to IFIFO. The output from the Corelet is aligned in the OFIFO and stored ins the memory module PMEM. The SFP module is not used in our Output Stationary model since the accumulation is done during execution, it will be described in Alpha5 of +Alpha part in detail.

Part 6: +Alpha

Alpha1: Resnet20 Training Mapping

We used the methodology in the **Figure 3** to modify resnet20.

Accuracy: 89.080%; PSUM recovered error: 0.0398. (Performance is not as good as VGG16)

<u>Analysis</u>

1. Perhaps because the ResNet20 model is modified too much. (We have to configure down sample layer of adjacent blocks to match input channels and output channels.)

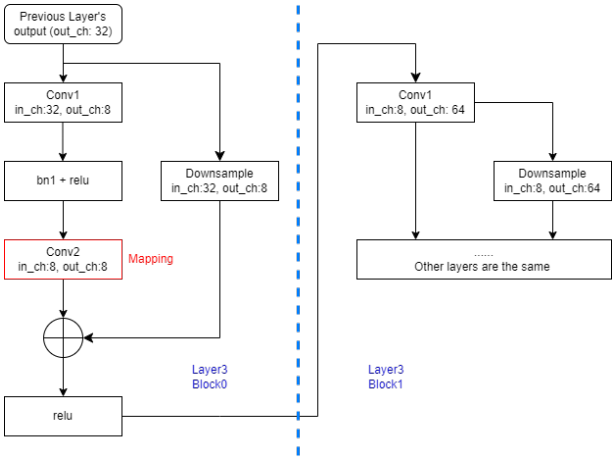

Figure 3. Method of modifying resnet20

2. The layer we choose to map has more data than that in VGG16 (in VGG16, a single channel is 4*4; in ResNet20, a single channel is 8*8). The calculation function is sum instead of mean, so the final error becomes larger.

Alpha2: Pruning

Pruning in Software

The results are shown in the **Table 3**.

Table 3. Pruning results in software

	Mapping layer 80% sparsity	All layers 80% sparsity	Mapping layer 40% sparsity
Unstructured pruning	89%	88%	
Structured pruning			73%

Analysis

1. Structured pruning is unable to achieve high accuracy with high sparsity, the reason might be structured pruning will cause the entire channel to disappear (when dim=0), resulting in a large impact on model.

2. Pruning on mapping layer will influence more, since the input and output channels of mapping layer are reduced to 8. Relatively speaking, the scale of parameters in the mapping layer is already small, which means the importance of each parameter will increase.

Pruning in Hardware

The pruning operation is executed based on the selected pruning_version, as detailed below:

Element-wise Pruning: Each element of the input vector is compared with unit_thres. If the absolute value exceeds the threshold, the element is retained; otherwise, it is set to zero.

Partial Sum Pruning: The vector is split into two groups of pruning_num elements each. The norms (sums of absolute values) for both groups are compared against unit_thres × pruning_num. If the condition is met, the group is retained; otherwise, it is zeroed out.

Global Norm Pruning: Computes the norm across all input elements. Retains the entire input vector if the global norm exceeds unit_thres × col; otherwise, sets it to zero.

Default Pruning: No pruning is applied, and the input is passed through as is Code Highlights.

The results are shown in the **Figure 4**.

Figure 4. Pruning results in hardware

Alpha3: Huffman

The methodology of generating Huffman coding is shown in the **Figure 5**.

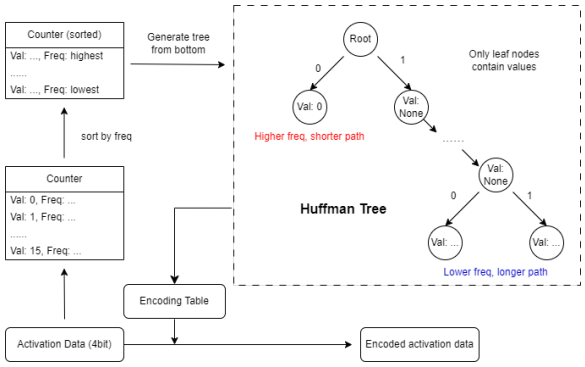

Figure 5. Method of Huffman coding

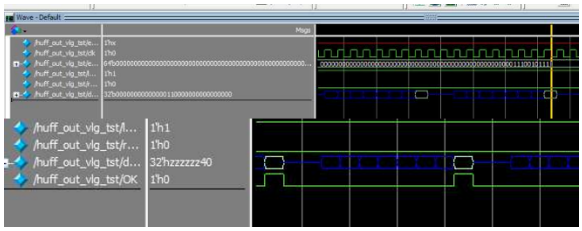

Figure 6. Results of Huffman decoder in hardware

And we designed the Huffman decoder in the hardware. The results are shown in the **Figure 6**, which are the same as the values of software.

Alpha4: Multi-Core Processor

The Multi-Core Processor architecture is illustrated in **Figure 7**. Activations and weights are first stored in XMEM. Then weights are transferred to IFIFO, activations are transferred to L0 and L1. When 'execute' is set to 1, Corelet1 and Corelet2 begin to work at the same time. Their outputs are aligned in the OFIFO and stored in the memory module PMEM. Note that the SFP module is also not used since this processor is for Output Stationary mode.

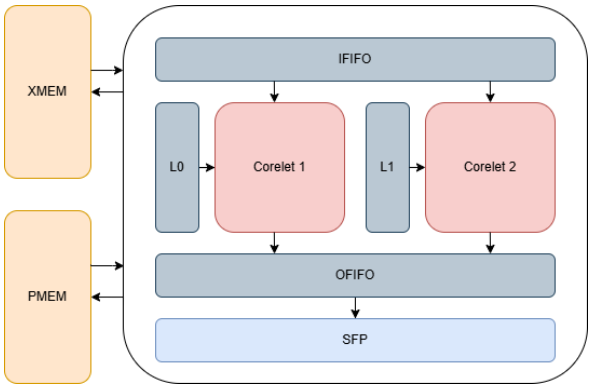

Figure 7. Multi-core processor architecture

Alpha5: Accumulation while Execution

For Output Stationary mode, we develop the Accumulation while Execution method so that we don't need SFP module anymore. The modified PE is illustrated in the **Figure 8**. The 'Sum' register is added to perform accumulation and store the partial sum value. The 'Counter' register is to track how many time accumulations are performed between different input channels. In our condition, input channel equals 3, so when the Counter reaches 3 it will judge that the accumulations are done and transfer the accumulated value to 'OS_Out' and set 'OS_Valid' to 1.

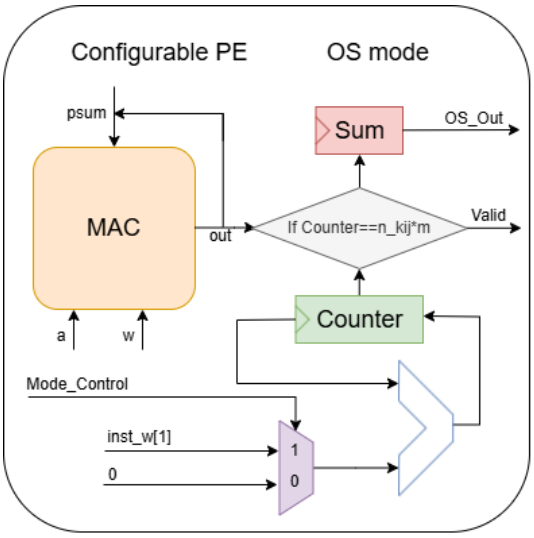

Figure 8. Architecture of modified PE