

ECE284 Fall 21 W5S2

Low-power VLSI Implementation for Machine Learning

Prof. Mingu Kang

UCSD Computer Engineering

Huffman Encoding

Your data: aabacdab

Frequency of a: 4

b: 2

c: 1

d: 1

4 symbol encoding

Assign a = 0

b = 10

c = 110

d = 111

aabacdab = 00100110111010

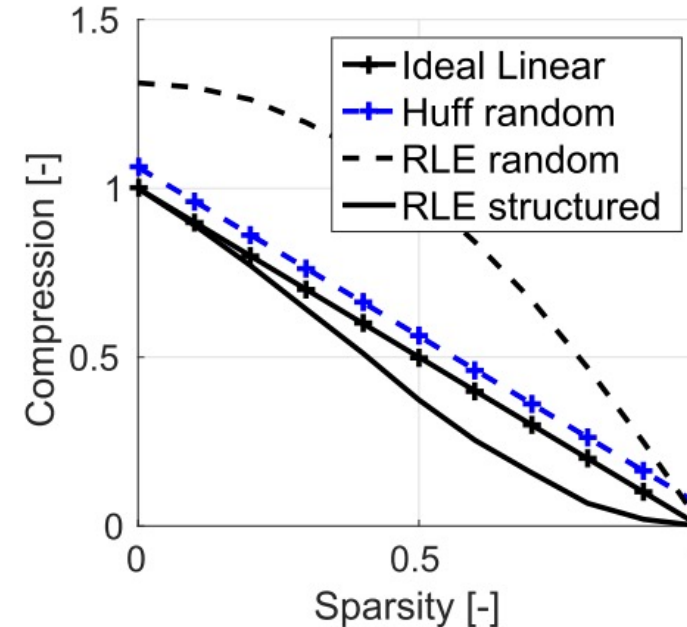
Decode => (0|0|10|0|110|111|0|10)

- Represent low number of bits for frequently happening symbol
- The encoding should satisfy “prefix rule: uniquely decodable rule”

Huffman Encoding Example

2-symbol Huffman encoding

- Zero: 1'b0
- Non zero: 17-b (1'b1, 16'b data)

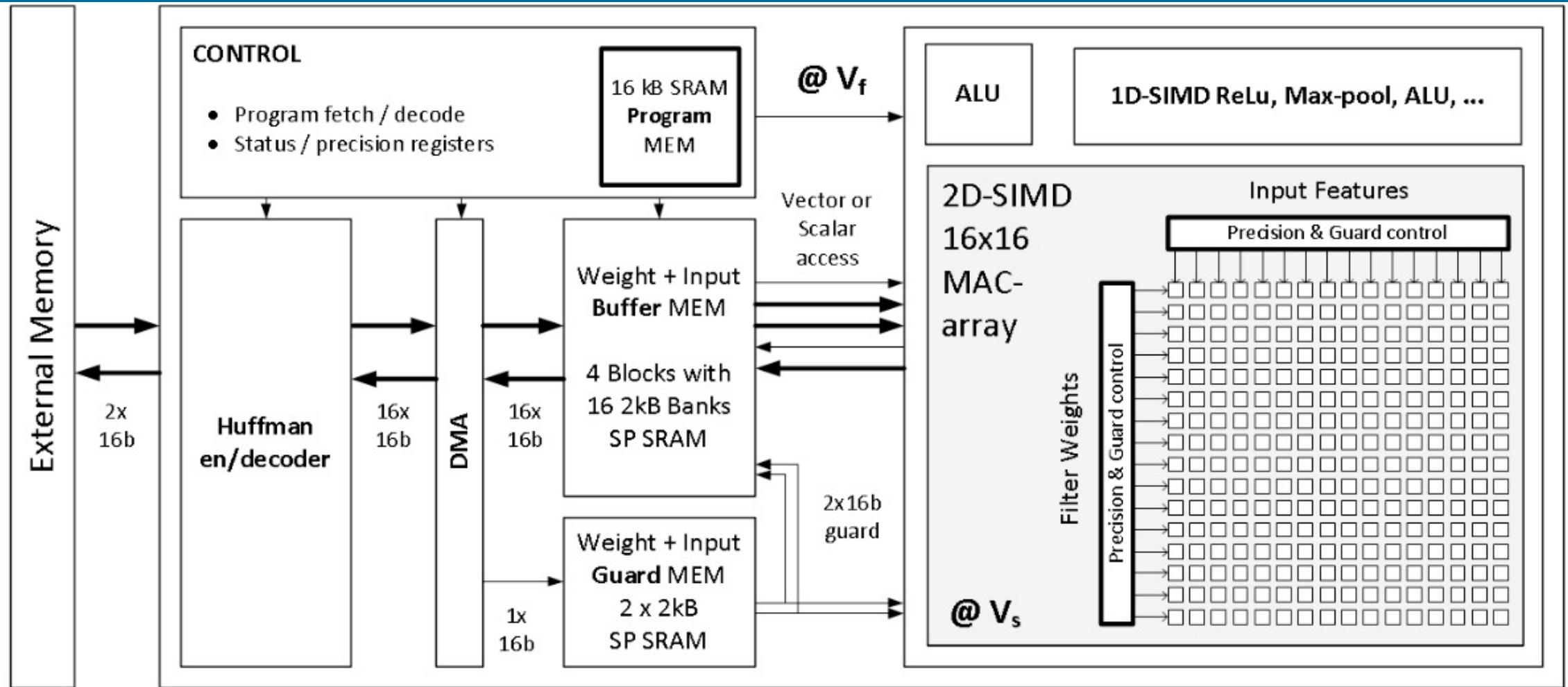


- Compressed data size, $C = (s \times (1/n) + (1 - s) \times ((n + 1)/n))$
where s : sparsity, n : word length (16)
- Random: the position of zero is random
- Structured: zeros are well clustered
- Huffman achieves a good linearity regardless of the zero position

[0101010101010101]
vs.
[0000000011111111]

In RLE vs. Huffman

Huffman Encoding-based Architecture Example

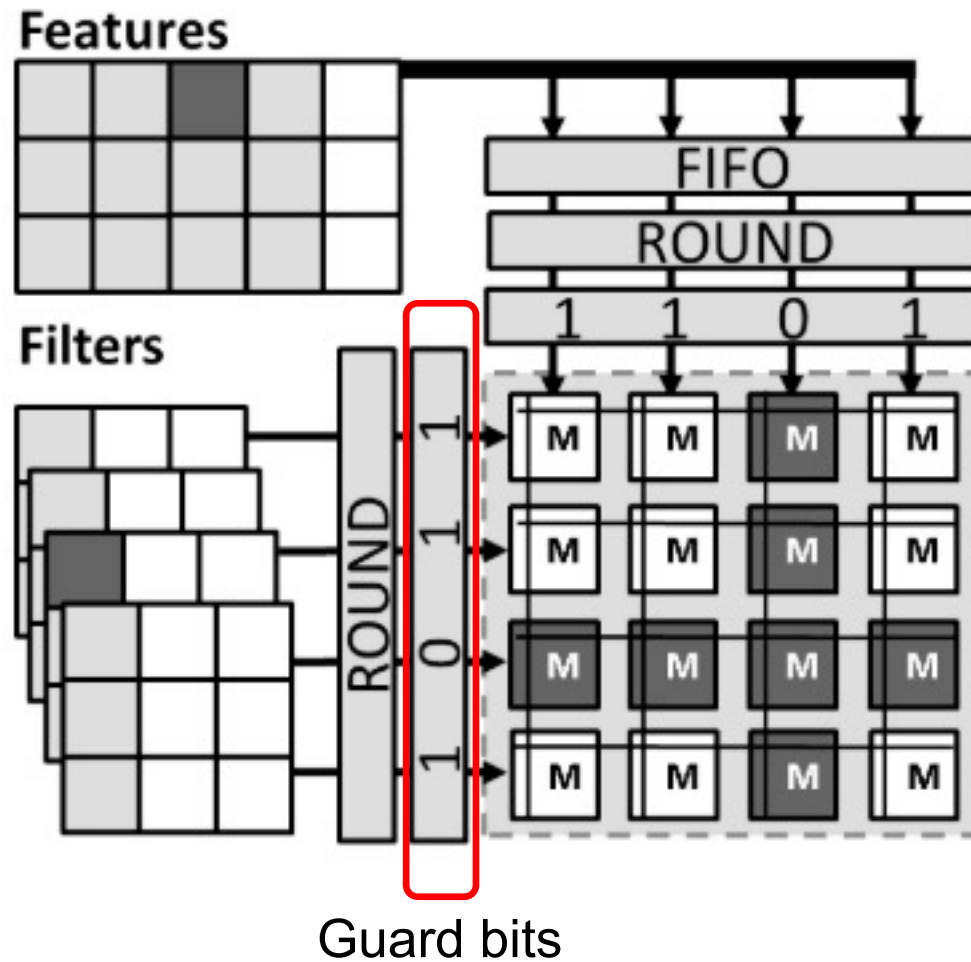


B. moon, "An energy-efficient precision-scalable convnet processor in 40-nm CMOS", JSSC17

- 16 words (each word is 16b) are fetched simultaneously.
- Check the guard (sparsity) bit first. If it is non-zero, then fetch. Otherwise, the memory bank is gated.

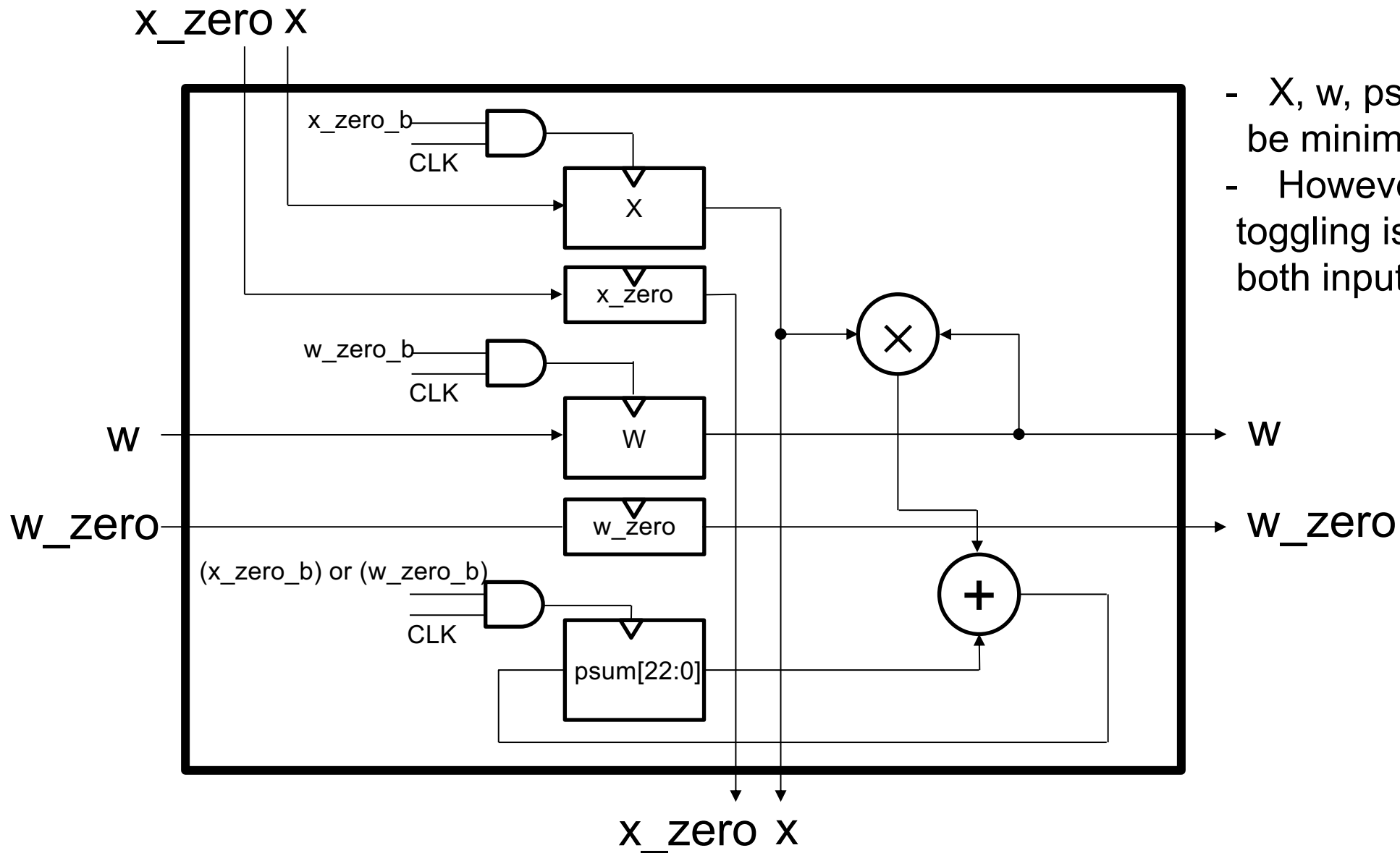
PE Design for Sparsity

Output stationary architecture



- Once the activation is zero, all the column is gated for one cycle
- Similarly, once the weight is zero, all the row is gated for one cycle

Limitation of Gating



- X, w, psum latch toggling can be minimized
- However, multiplier and acc. toggling is unavoidable unless both inputs are zero

Compressed Sparse Column (CSC) format

<https://matteding.github.io/2019/04/25/sparse-matrices/>

	0	1	2	3	4
0	8		2		
1			5		
2					
3					
4			7	1	2
5					
6				9	

© Matt Eding

CSC

Index pointer (IP)

0	1	1	4	6	7
---	---	---	---	---	---

Row index(RI)

0	0	1	4	4	6	4
---	---	---	---	---	---	---

Data

8	2	5	7	1	9	2
---	---	---	---	---	---	---

Required data size

$$= 2 * (\# \text{ of non zeros}) + \# \text{ of column} + 1$$

- Row index (RI) contains the non zero value's row numbers in the column
- In the n-th column, non zero datum's row index: $RI[IP[n]], \dots, RI[IP[n+1] - 1]$
- Difference between two consecutive values in index pointer (IP) = # of non zero elements in the column

Compressed Sparse Row (CSR) format

<https://matteding.github.io/2019/04/25/sparse-matrices/>

Col index

	0	1	2	3	4
0	8		2		
1			5		
2					
3					
4			7	1	2
5					
6				9	

© Matt Eding

CSR

Index pointer (IP)

0	2	3	3	3	6	6	7
---	---	---	---	---	---	---	---

Col index (CI)

0	2	2	2	3	4	3
---	---	---	---	---	---	---

Data

8	2	5	7	1	2	9
---	---	---	---	---	---	---

Required data size

$$= 2 * (\# \text{ of non zeros}) + \# \text{ of row} + 1$$

- col index contains the non zero value's row numbers in the row
- In the n-th row, non zero datum's column index : $CI[IP[n]], \dots CI[IP[n+1] - 1]$
- Difference between two consecutive values in IP = # of non zero elements in the row

[Example1] Unstructured Pruning for MNIST

- “model.conv1. weight” is removed from the named_parameters()
- weight_orig and weight_mask are added in the named_parameters()
- However, retraining is possible without any further modification
- Retraining recovers the accuracy significantly
- Saved file cannot be loaded on the model directly (because the weight is not in the named_parameter any more)
- Thus, run prune commands and then load the pruned saved file

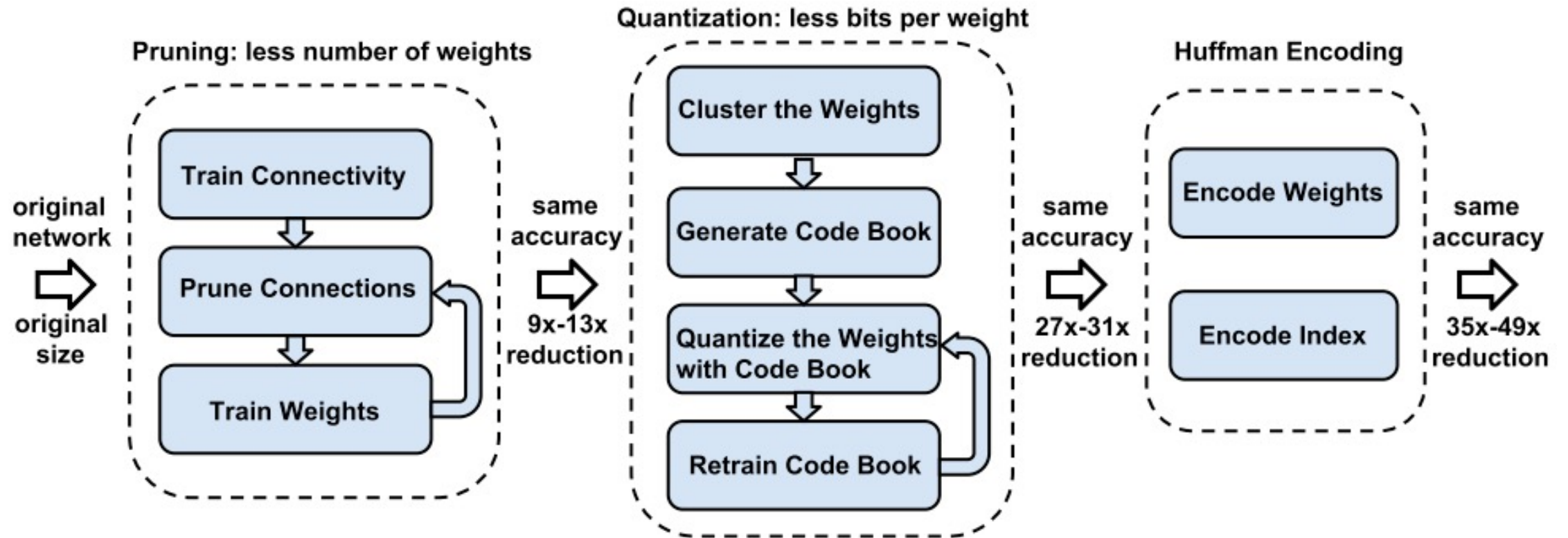
[Example2] Structured Pruning for MNIST

- Structured pruning removes “dim” dimension in the weight matrix
- Also, it employs L-n norm. If $n = 1$, it uses L1 norm. This is why the function name is “ln_structured”
- Accuracy drop is more significant


[HW_prob1] Pruning for Quantized VGGNet

- Open [HW6_prob1]_VGG16_Quantization_aware_train_with_pruning.ipynb
- Call your 4-bit quantization-aware trained VGGNet checkpoint
- Apply unstructured pruning with 90% sparsity for the weights for all the conv layers
- Check your accuracy
- Retrain to recover the accuracy as much as you can
- Check how much accuracy you can achieve
- Now, iterate the above process with structured pruning

ICLR16: Deep Compression



Weight Clustering and Sharing

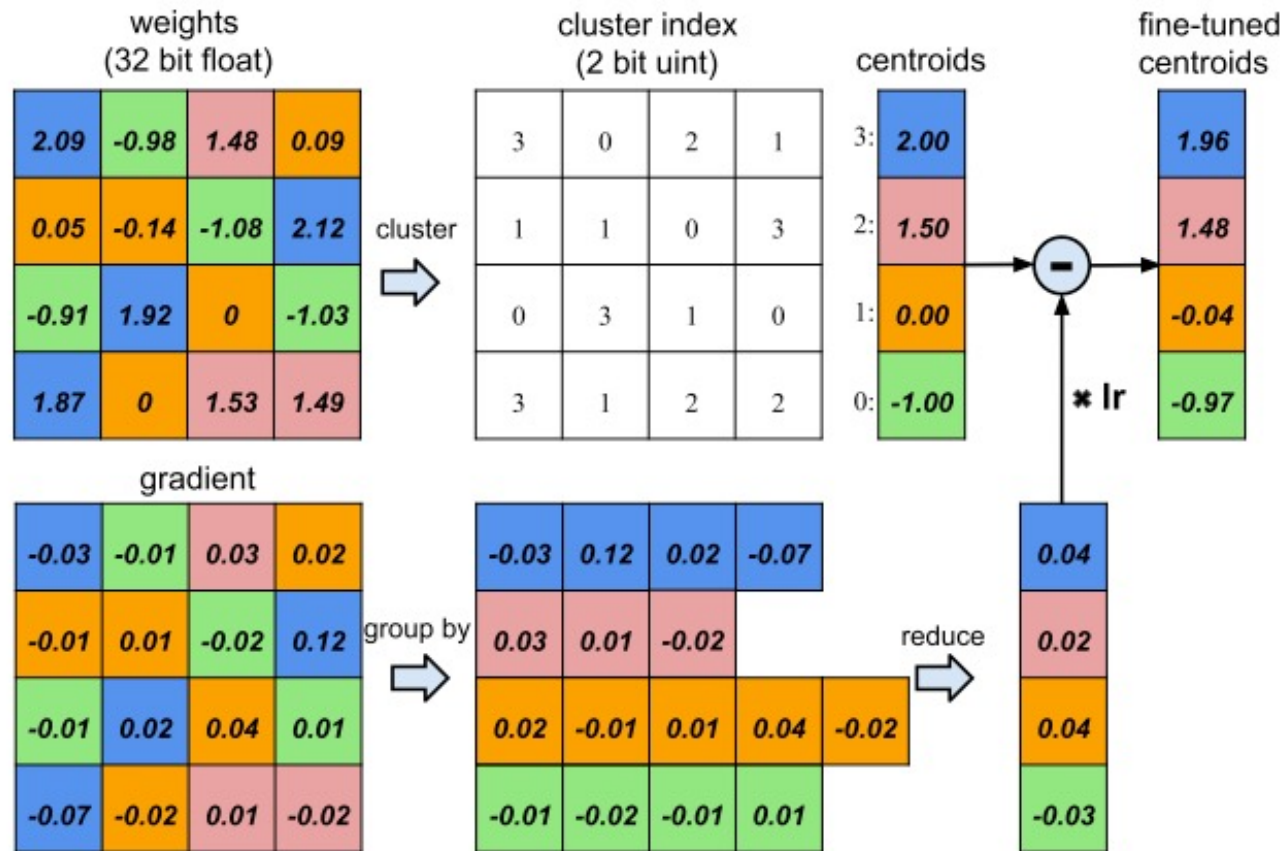
n original weights $W = \{w_1, w_2, \dots, w_n\}$  k clusters $C = \{c_1, c_2, \dots, c_k\}$
 $n \gg k$

$$\arg \min_C \sum_{i=1}^k \sum_{w \in c_i} |w - c_i|^2$$

Codeword generation

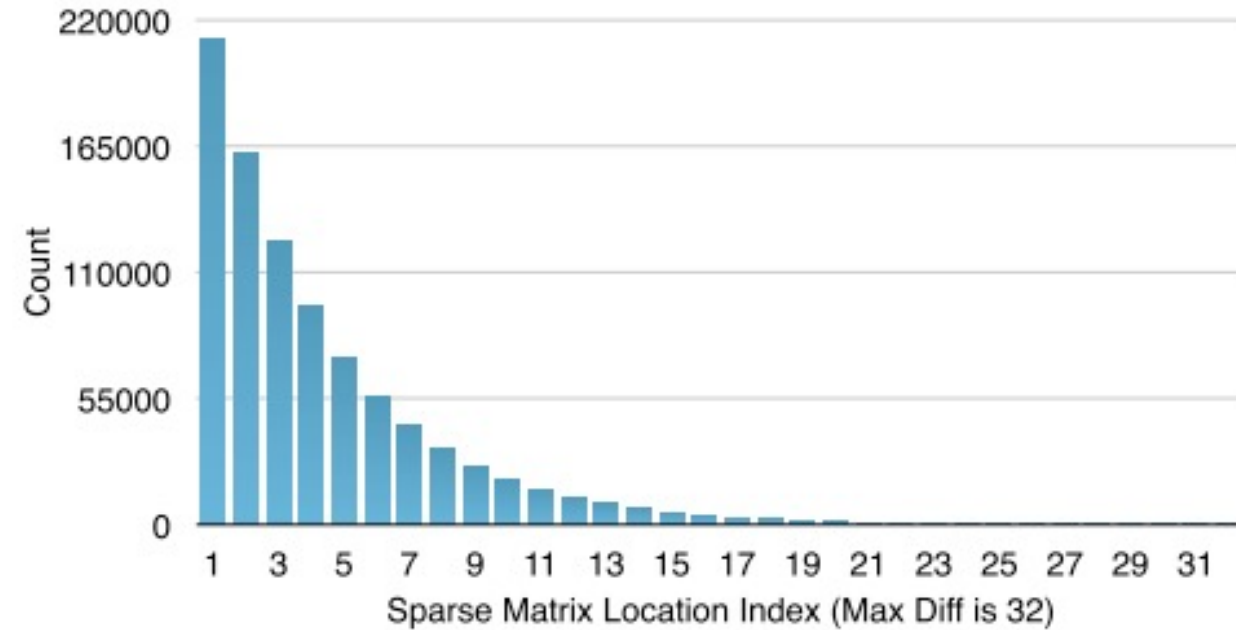
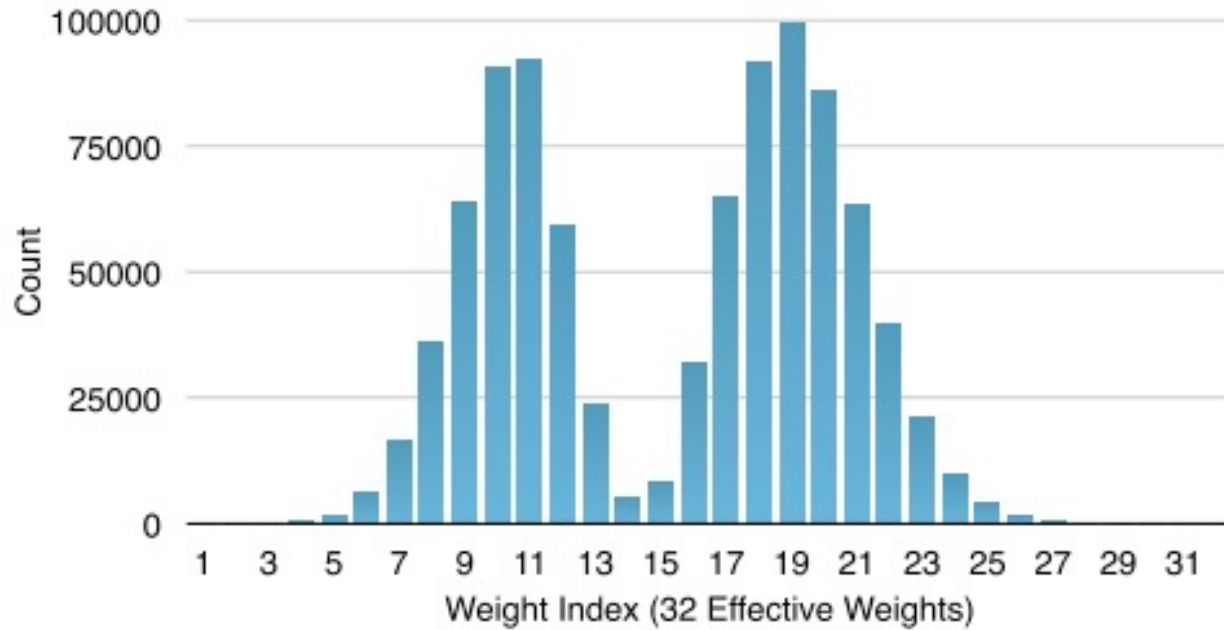
- 256 shared weights (8-bits) for CONV layers
- 32 shared weights (5-bits) for FC layer

Retrain Code Book



- 4 code words (by color) for weight.
- The gradients are summed up based on the color, then multiplied with learning rate and updated.

Huffman Encoding



Fully connected layer weights

- Assigned lower bit index for the frequently used code
- 20-30% additional storage savings

Benefit Summary

- Pruning: reduces the number of connections by 9× to 13×
- Quantization: reduces the number of bits that represent each connection from 32 to 5
- On the ImageNet dataset, without loss of accuracy,
 - the storage size for AlexNet reduced by 35×, from 240MB to 6.9MB,.
 - the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy.
- Allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory