上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS4302-01
Parallel and Distributed Computing

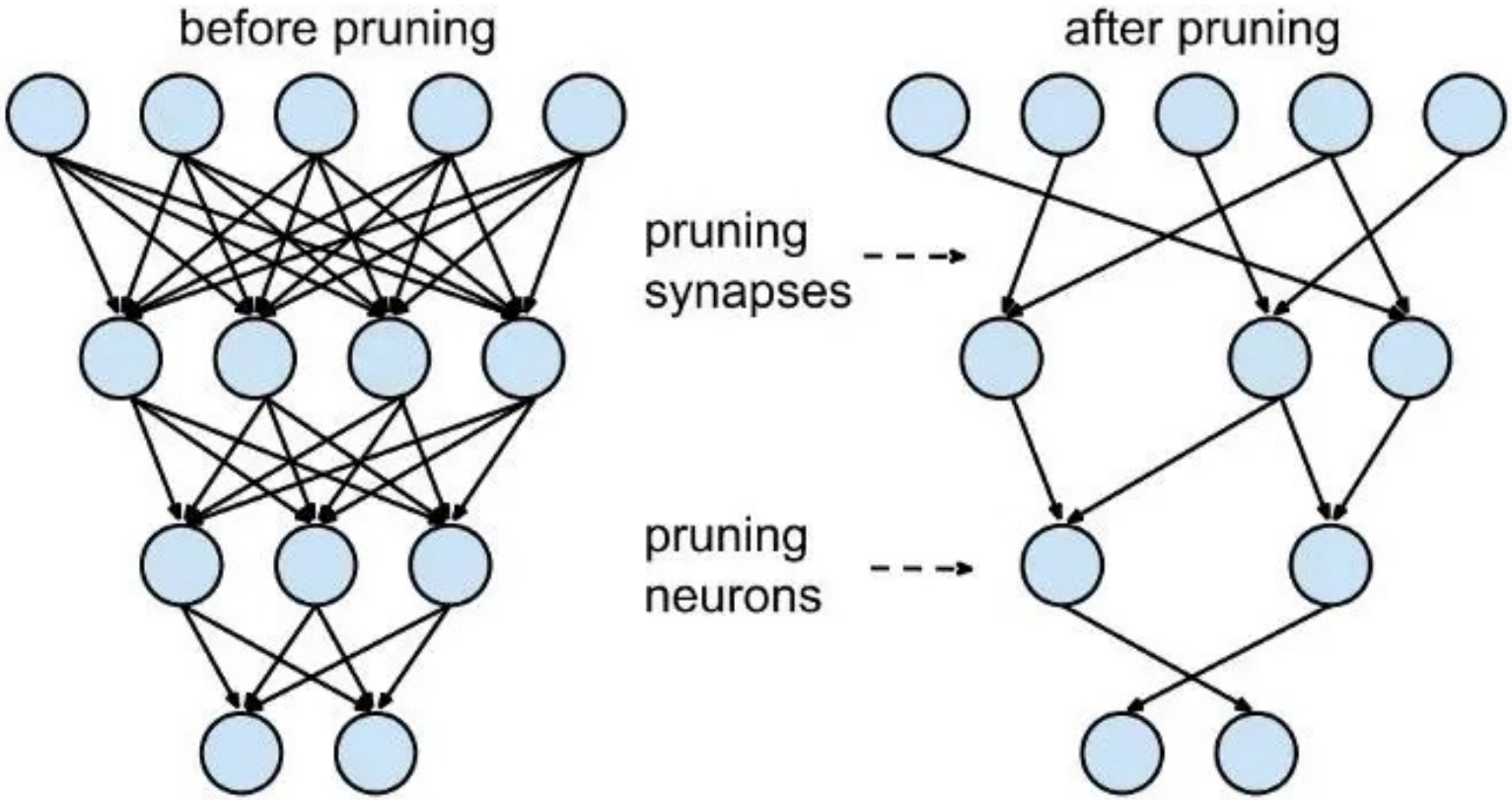
Lecture 11 Efficient DNN Processing

Zhuoran Song

2023/10/31

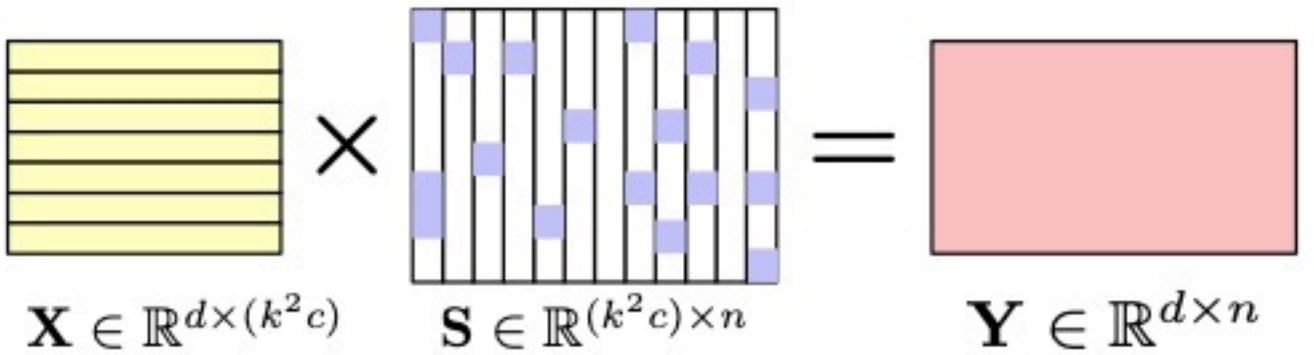
Pruning

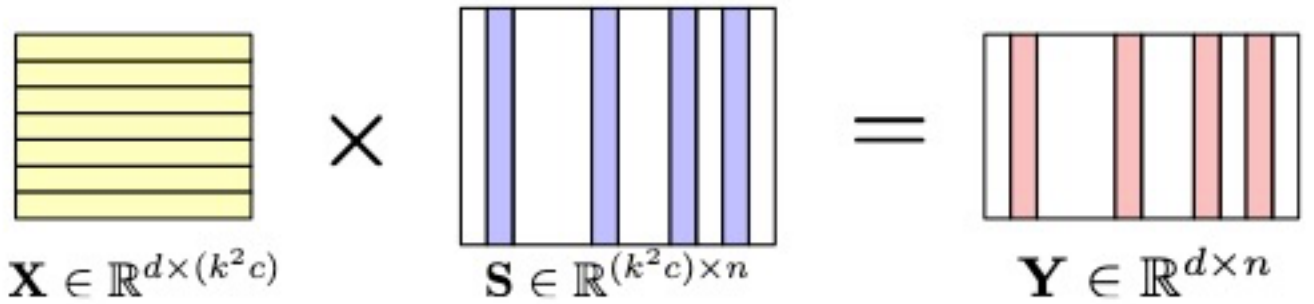
- **Make the synapses to be zeros**



Pruning

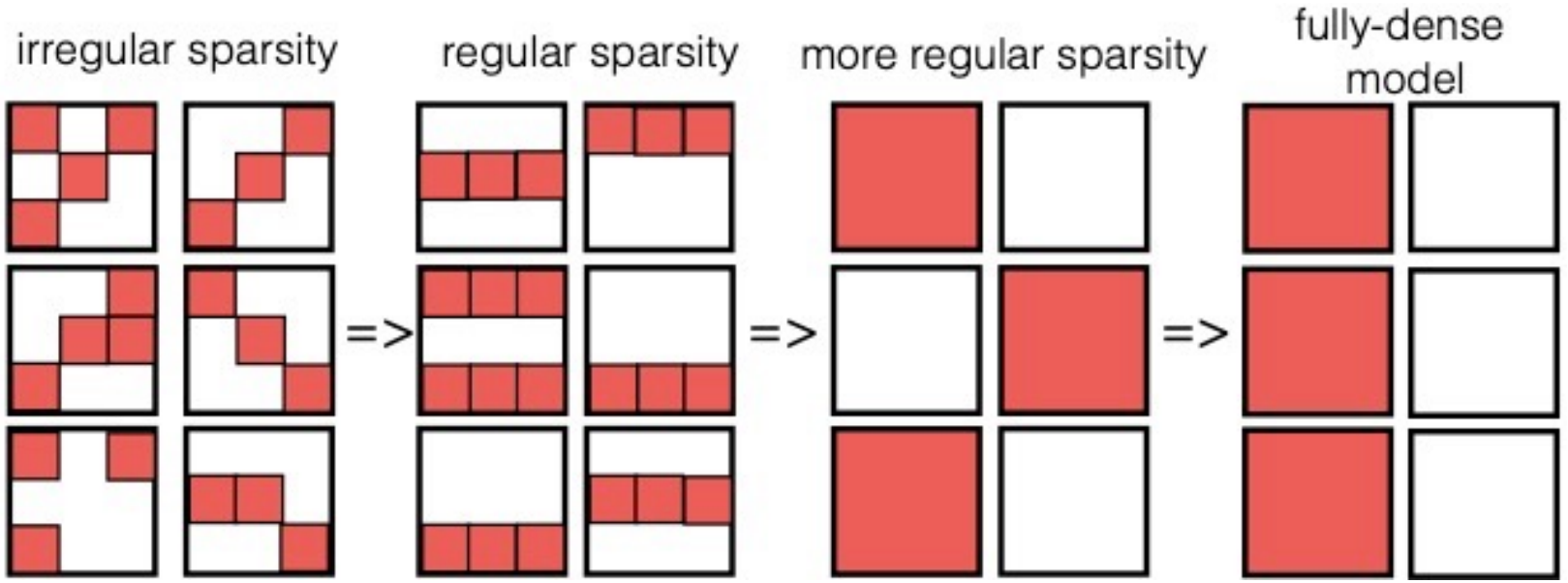
- **Compression: compressed sparse format for storage**
- **Potential acceleration: sparse matrix multiplication algorithm**
- **Random Sparsity vs. Structured Sparsity**





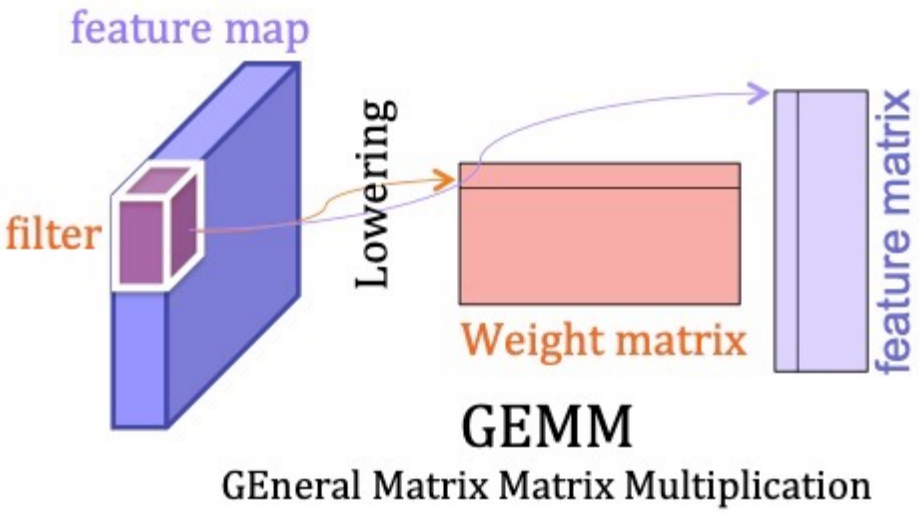
Hardware-friendly Sparsity

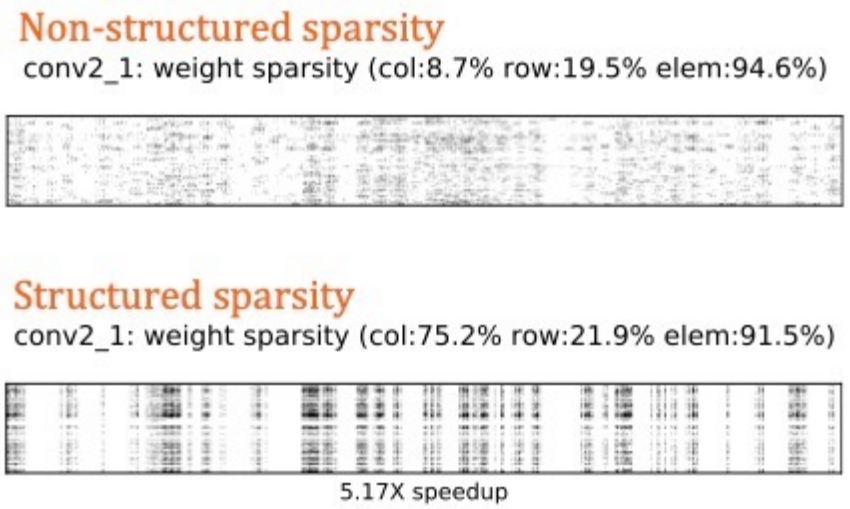
- **4-type of sparsity**



Structural Sparsity Learning

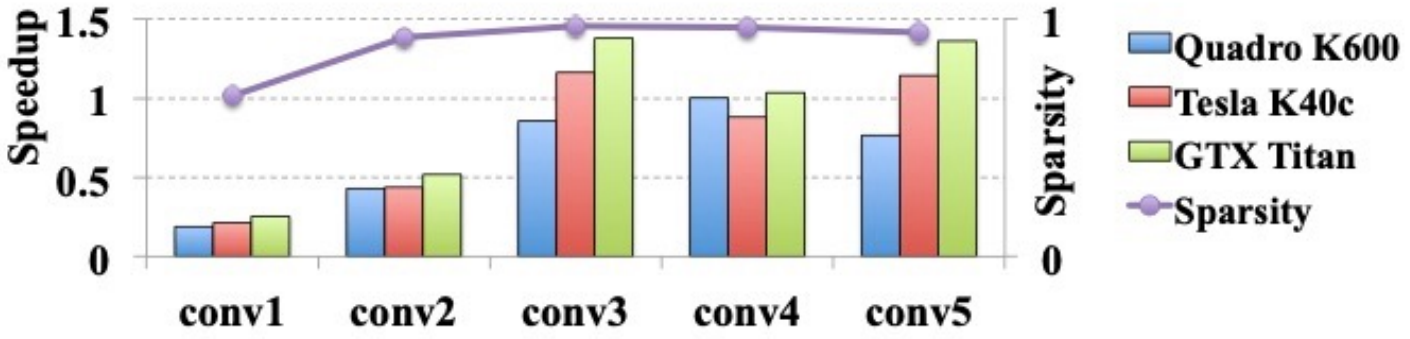
- **Removing rows/columns in GEMM (row/column-wise sparsity)**



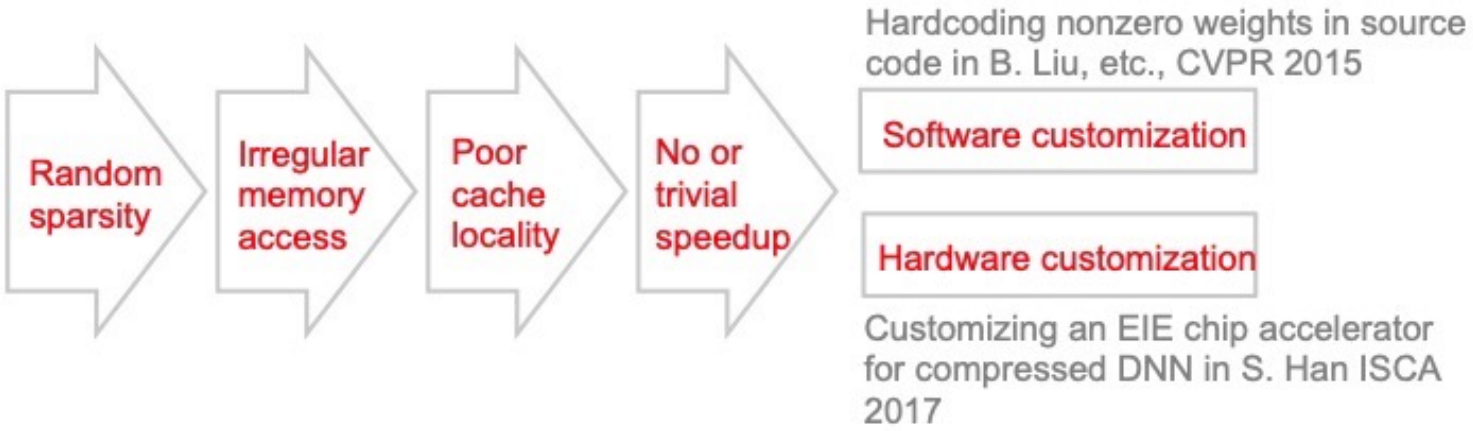


Pruning-Speedup

- **Random sparsity, theoretical speedup ≠ practical speedup**

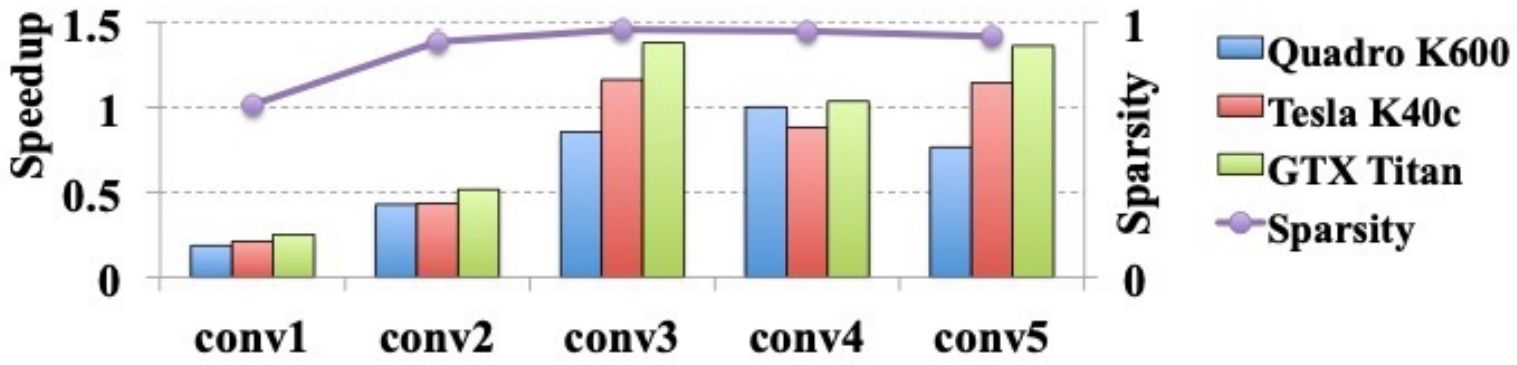


Forwarding speedups of AlexNet on GPU platforms and the sparsity. Baseline is GEMM of cuBLAS. The sparse matrixes are stored in the format of Compressed Sparse Row (CSR) and accelerated by cuSPARSE.

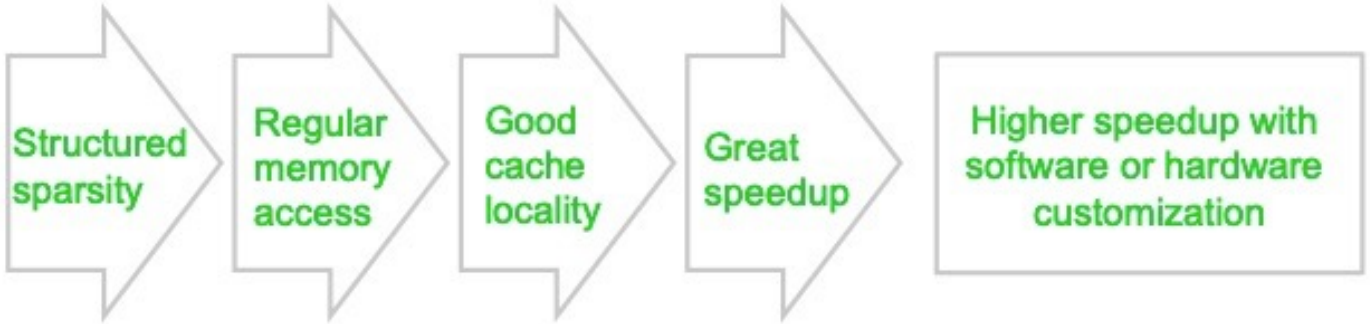


Pruning-Speedup

- **Structured sparsity**



Forwarding speedups of AlexNet on GPU platforms and the sparsity. Baseline is GEMM of cuBLAS. The sparse matrixes are stored in the format of Compressed Sparse Row (CSR) and accelerated by cuSPARSE.



Naive Implementation for Sparsity

X

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

*

W

0
0
4
8

Algorithm Sparse Convlution Naive 1

1: **for all** w[i] **do**
2: **if** w[i] = 0 **then**
3: Continue;
4: **end if**
5: output feature map $Y \leftarrow X \times w$[i];
6: **end for**

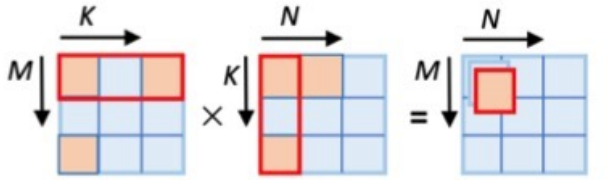
BAD implementation for Pipeline!

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Hardware Implementation for Sparsity

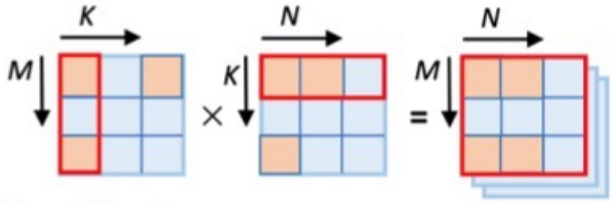
Inner-product dataflow

```
for m in 0..M
 for n in 0..N
  for k in 0..K
   C[m,n] += A[m,k] * B[k,n]
```



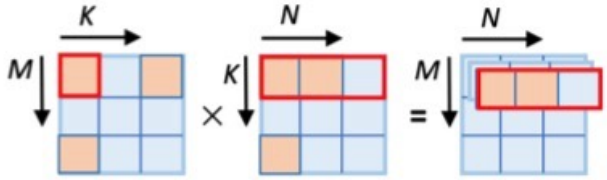
Outer-product dataflow

```
for k in 0..K
 for m in 0..M
  for n in 0..N
   C[m,n] += A[m,k] * B[k,n]
```



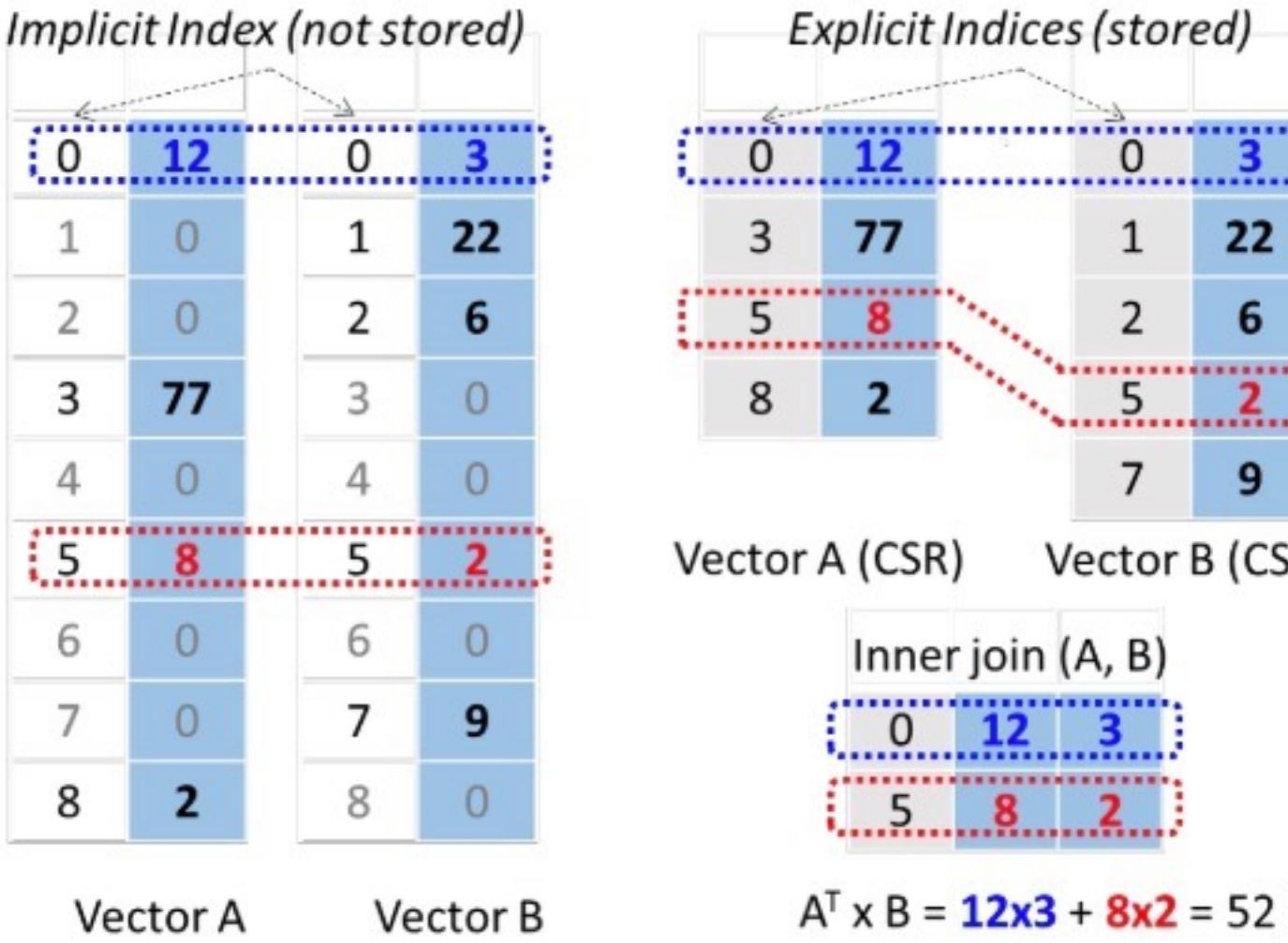
Row-based dataflow

```
for m in 0..M
 for k in 0..K
  for n in 0..N
   C[m,n] += A[m,k] * B[k,n]
```

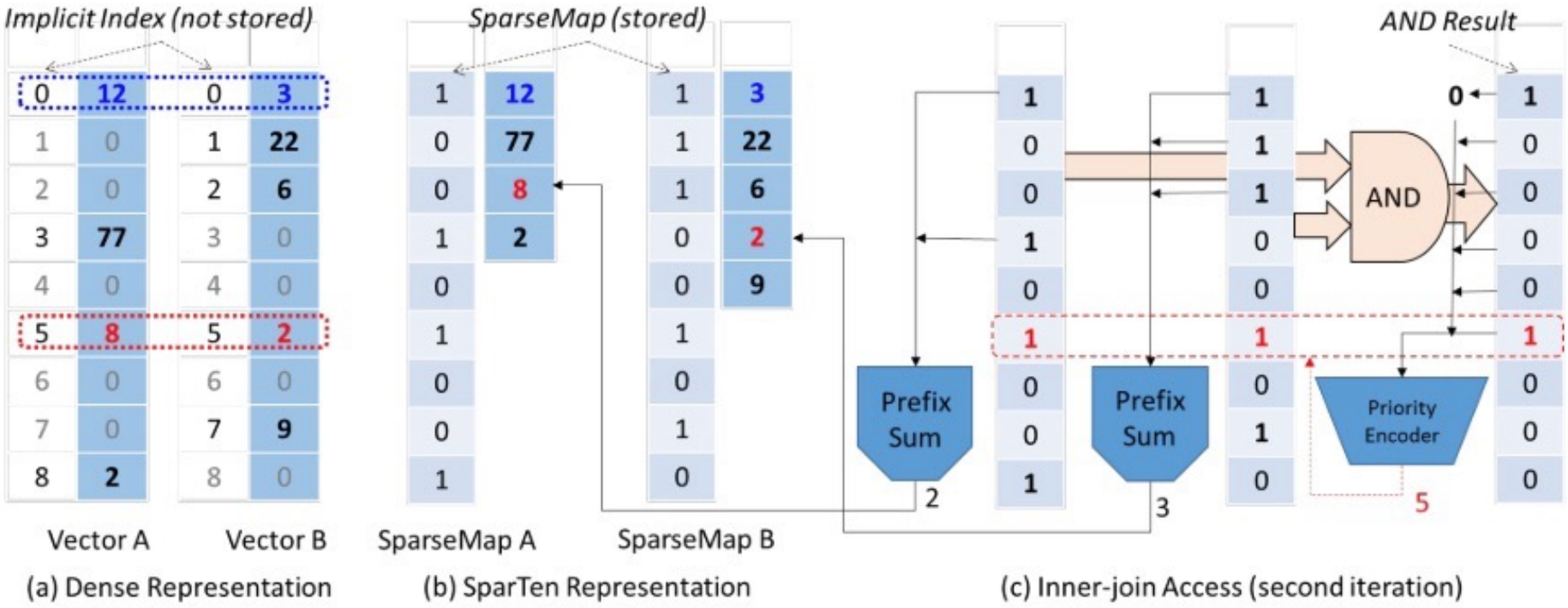


	InP	OutP	ROW
Input reuse (B)	Poor	Excellent	Poor
Output reuse (C)	Excellent	Poor	Good
Index intersection	Inefficient	Efficient	Efficient
Psum granularity	Scalar	Matrix	Vector

Inner Product Accelerator



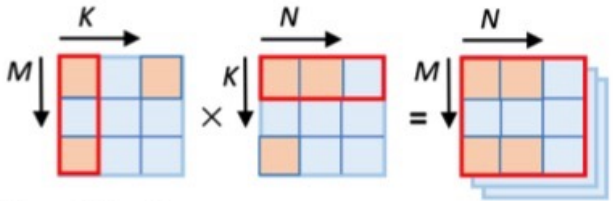
Inner Product Accelerator

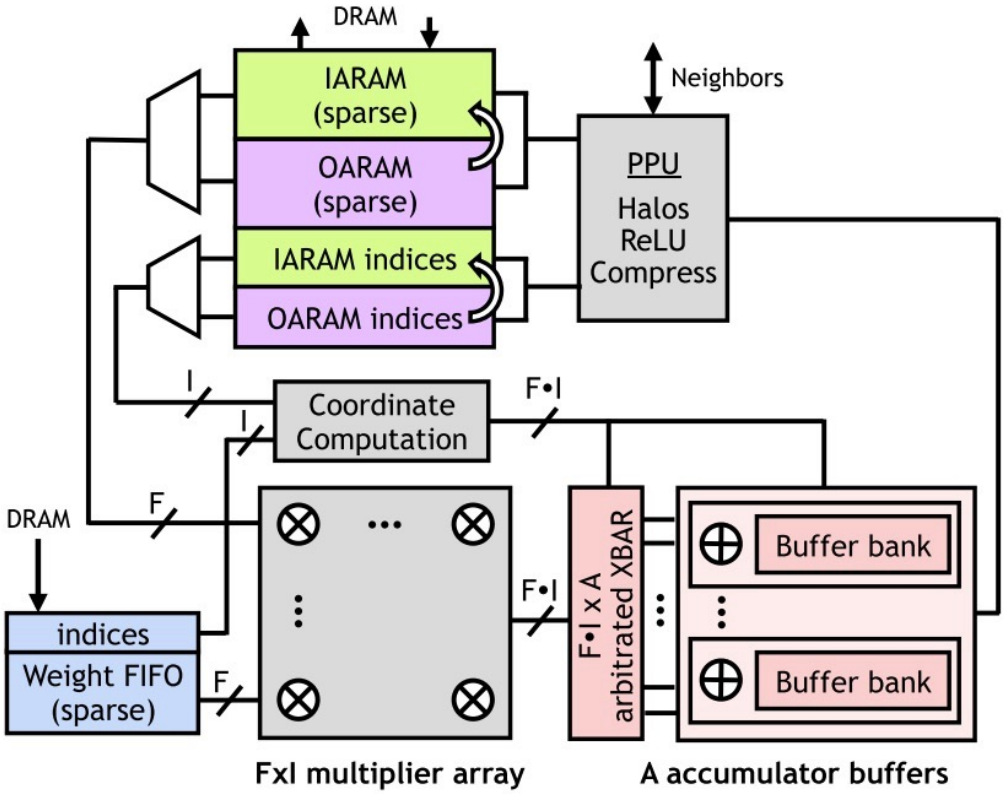


Outer Product Accelerator

Outer-product dataflow

```
for k in 0..K
 for m in 0..M
  for n in 0..N
   C[m,n] += A[m,k] * B[k,n]
```





SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks (ISCA'17)

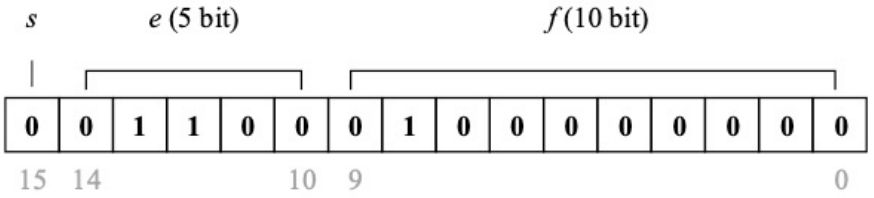
Floating point number

- Scientific notation: 6.6254×10^{-27}
- A normalized number of certain accuracy (e.g. 6.6254 is called the mantissa)
- -27 is called the exponent
- Floating Point Numbers can have multiple forms, e.g.

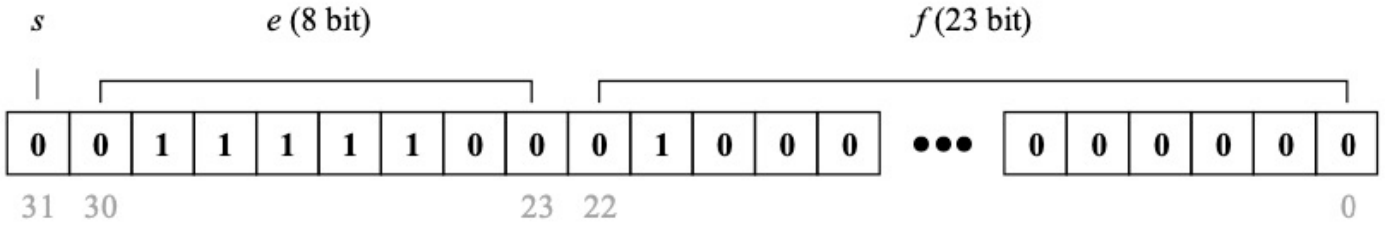
$$
\begin{aligned}
0.232 \times 10^4 &= 2.32 \times 10^3 \\
&= 23.2 \times 10^2 \\
&= 2320. \times 10^0 \\
&= 232000. \times 10^{-2}
\end{aligned}
$$

- It is desirable for each number to have a unique representation => Normalized Form
- We normalize Mantissa's in the Range [1..*R*), where R is the Base, e.g.:
  - [1..2) for BINARY
  - [1..10) for DECIMAL

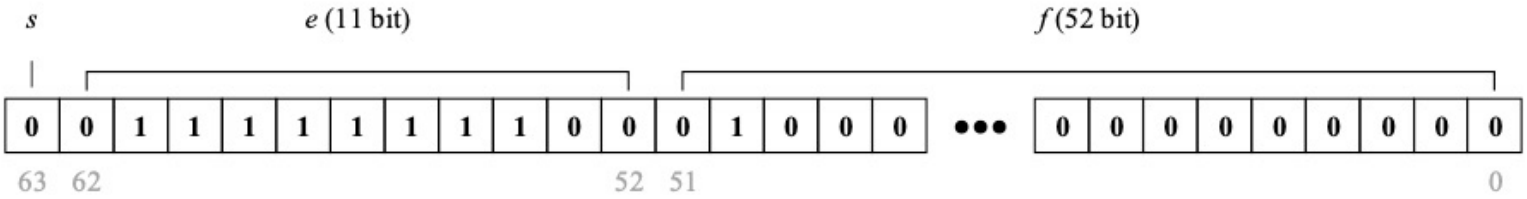
Floating point number



（a）FP16



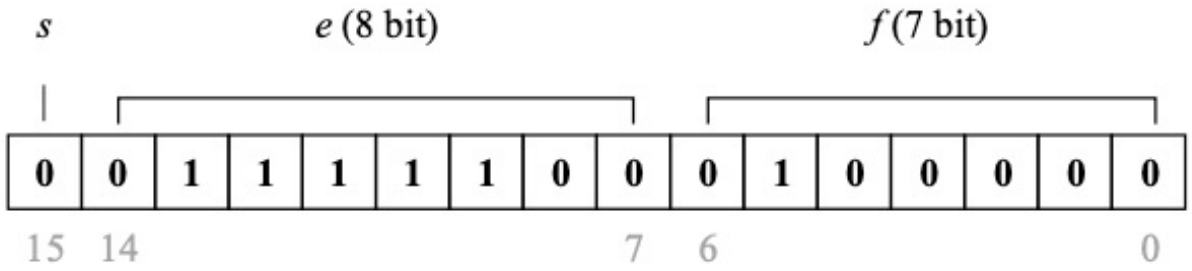
（b）FP32



（c）FP64

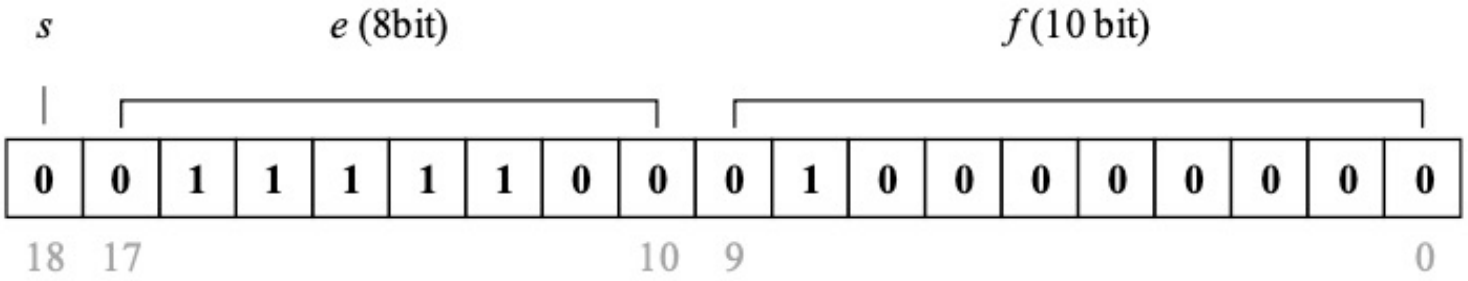
Google's BP16

- **Exponent is more important than mantissa**
- **Better for DNN training, high accuracy**
- **Applied to Nvidia GPU and Google's TPU**



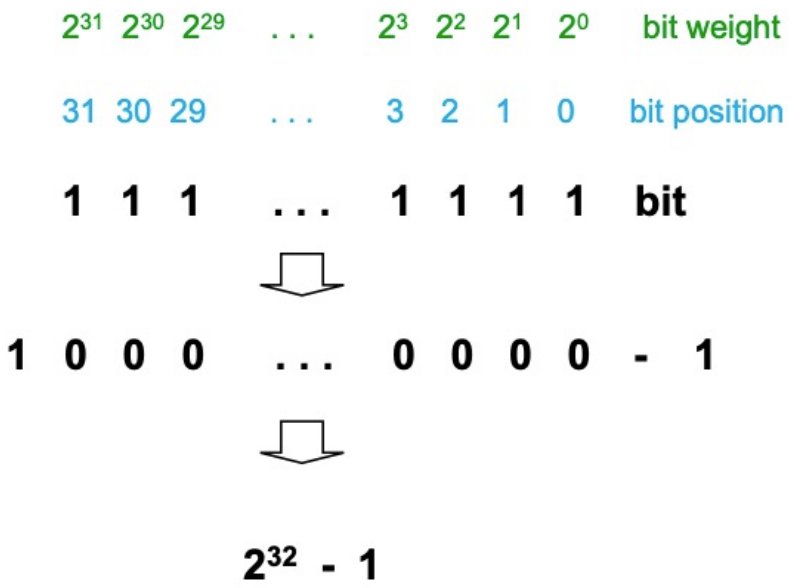
Nvidia’ TF32

- 8 bit exponent (same as FP32)
- 10 bit mantissa (same as FP16)
- Applied to Nvidia GPU, Ampere architecture
  - 156TFLOPS



Binary Representation

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	$2^{32} - 4$
0xFFFFFFFD	1…1101	$2^{32} - 3$
0xFFFFFFFE	1…1110	$2^{32} - 2$
0xFFFFFFFF	1…1111	$2^{32} - 1$



Quantization

- **Reduce storage cost**
- **Reduce computation complexity**

2.3	-1.1	-0.1
-0.2	2.1	-1.0
2.0	-1.2	0.1
2.1	0.1	-1.1

→

2.0	-1.0	0
0	2.0	-1.0
2.0	-1.0	0
2.0	0	-1.0

$$A \times W = Z$$

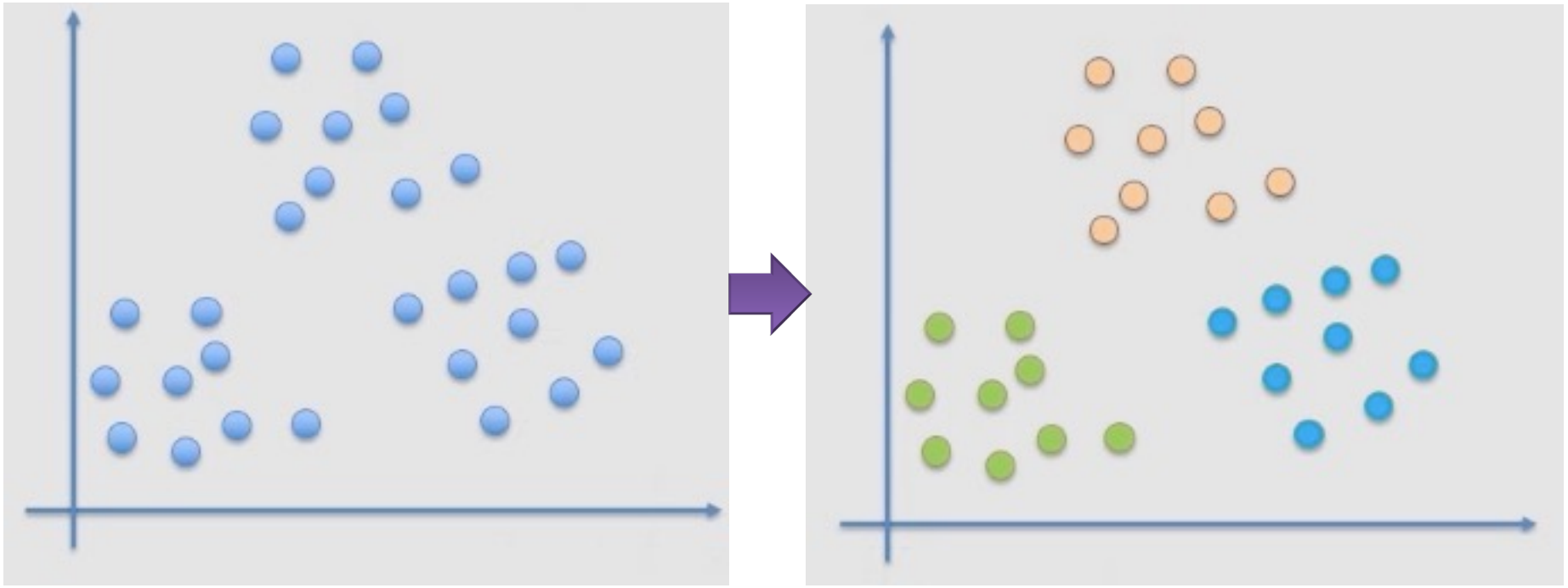
$$A = A_Q \times a + b$$

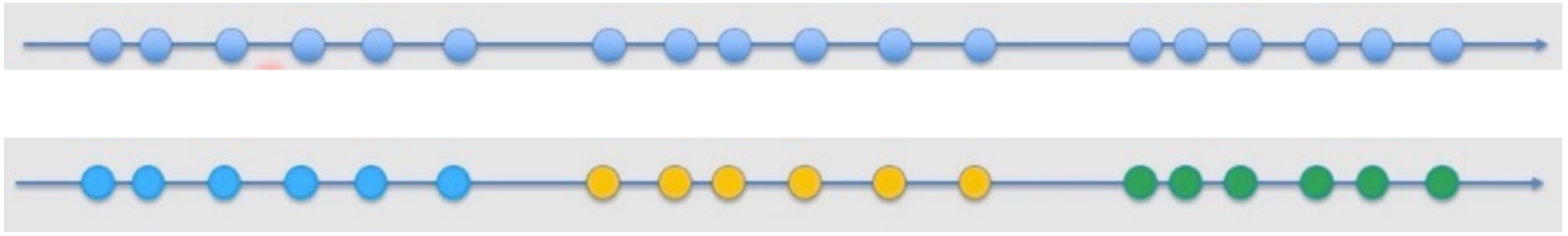
$$W = W_Q \times c + d$$

$$Z = (A_Q \times a + b) \times (W_Q \times c + d)$$

$$= acA_QW_Q + adA_Q + bcW_Q + bd$$

K-means





Linear Quantization

Quantization:

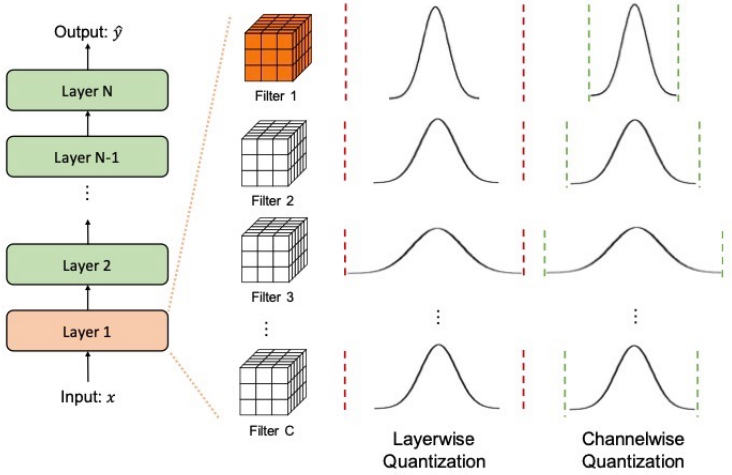
$$Q(r) = \text{Int}(r/S) - Z$$

Dequantization:

$$\hat{r} = S(Q(r) + Z)$$

Granularity:

- Layerwise
- Groupwise
- Channelwise



Scale

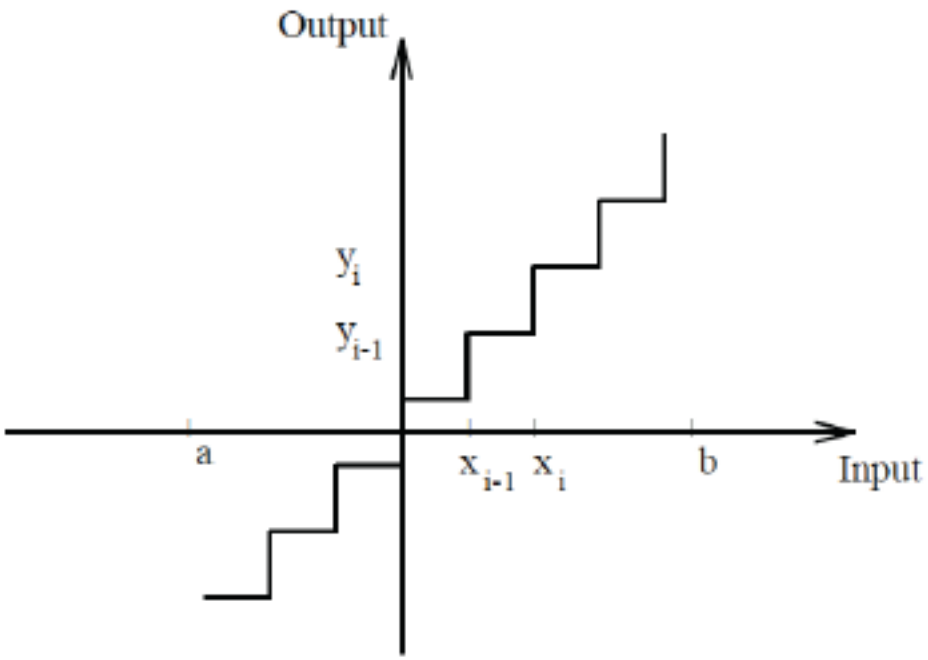
$$scale = \frac{(max_{val} - min_{val})}{(max_q - min_q)}$$

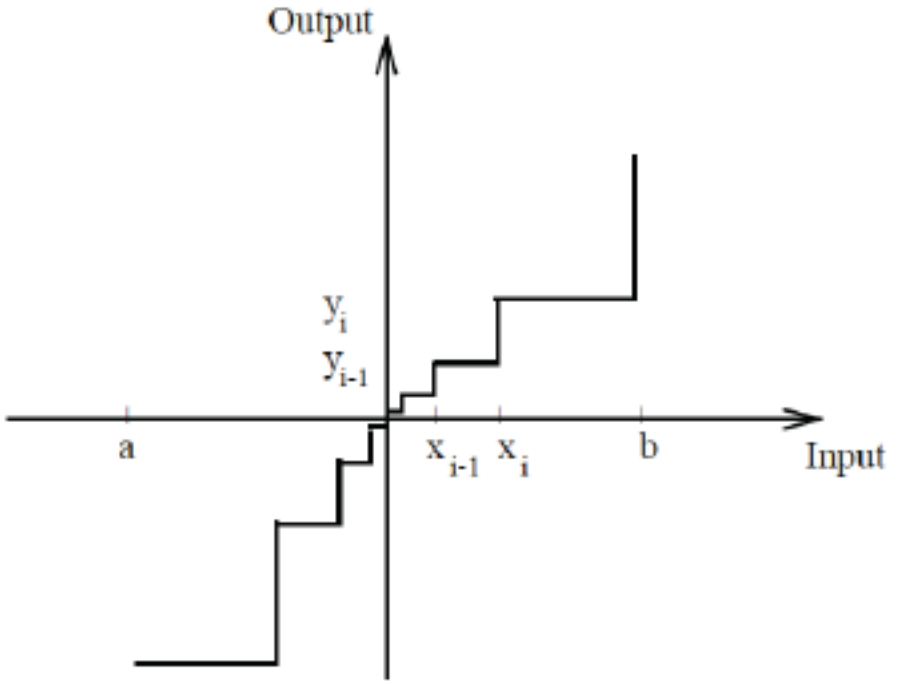
Zero point

$$zero\ point = 0$$

Uniform vs. Non-Uniform

- Real values in the continuous domain r are mapped into discrete
- Uniform quantization: distances between quantized values are the same
- Non-uniform quantization: distances between quantized values can vary

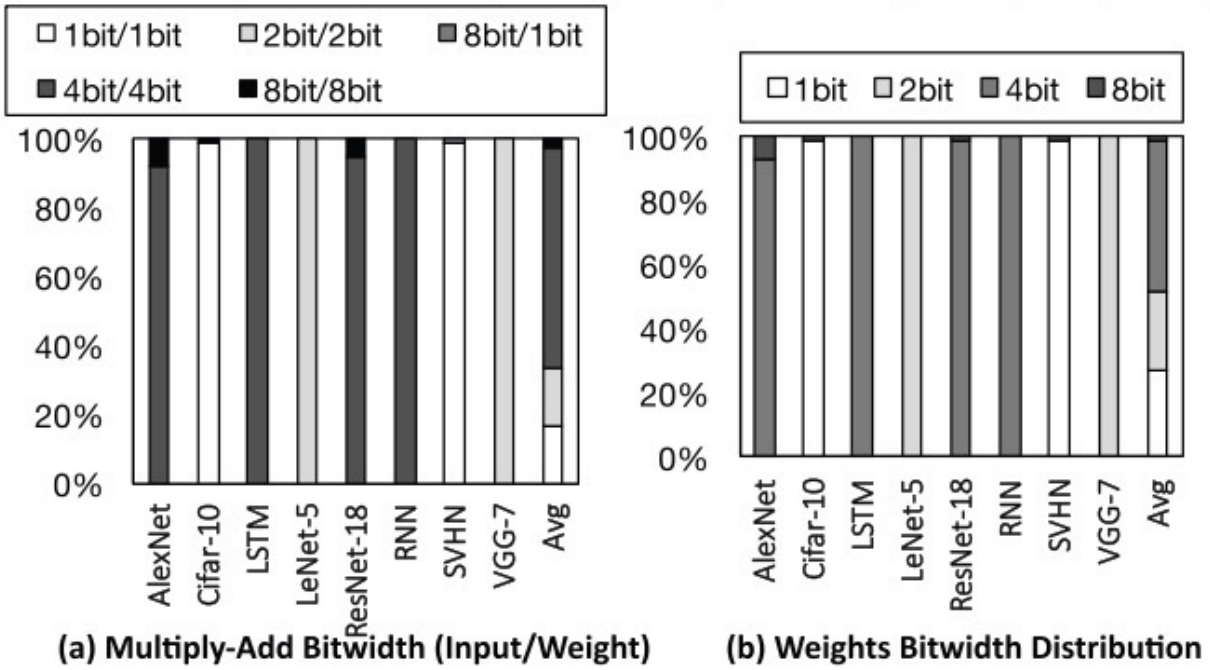




Quantization Accelerator

- **Bitwidth variation across real-world DNNs**

DNN	AlexNet	Cifar-10	LSTM	LeNet-5	ResNet-18	RNN	SVHN	VGG-7
% Multiply-Add	99.8	99.8	99.9	99.4	99.9	99.9	99.8	99.5

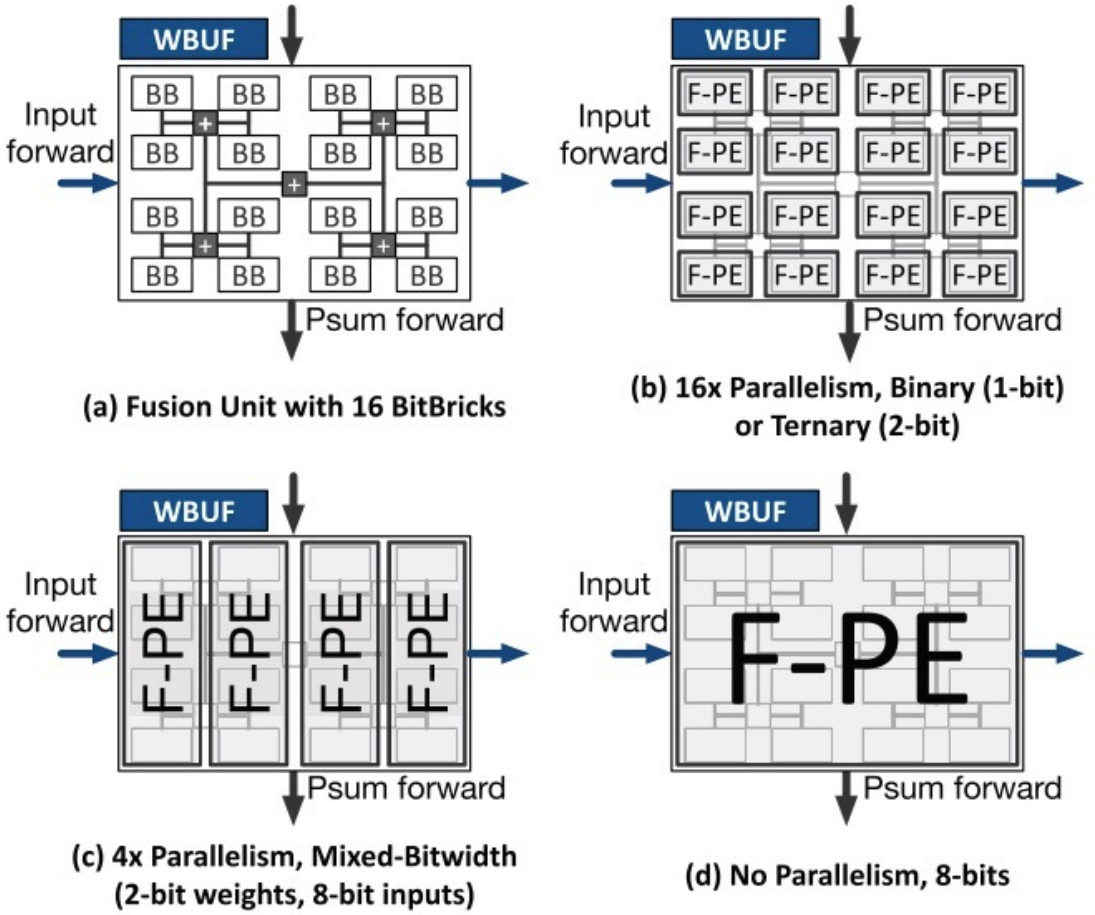


(a) Multiply-Add Bitwidth (Input/Weight) (b) Weights Bitwidth Distribution

Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks (ISCA'18)

Quantization Accelerator

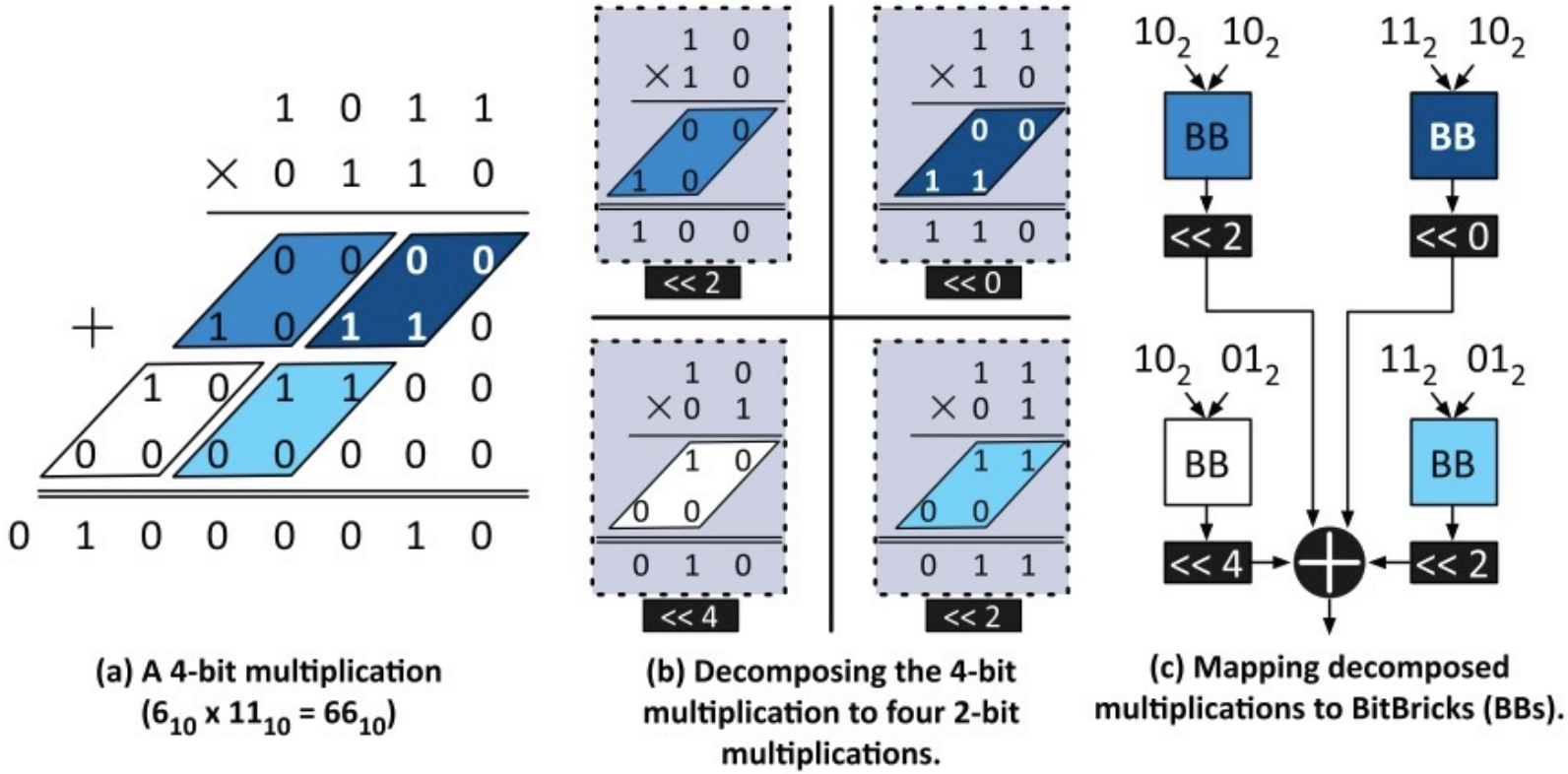
- **Dynamic composition of BitBricks (BBs) in a Fusion Unit to construct Fused Processing Engines (Fused-PE)**



Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks (ISCA'18)

Quantization Accelerator

- **Using BitBricks to execute 2-bit and 4-bit multiplications**



(a) A 4-bit multiplication (6_{10} x 11_{10} = 66_{10})

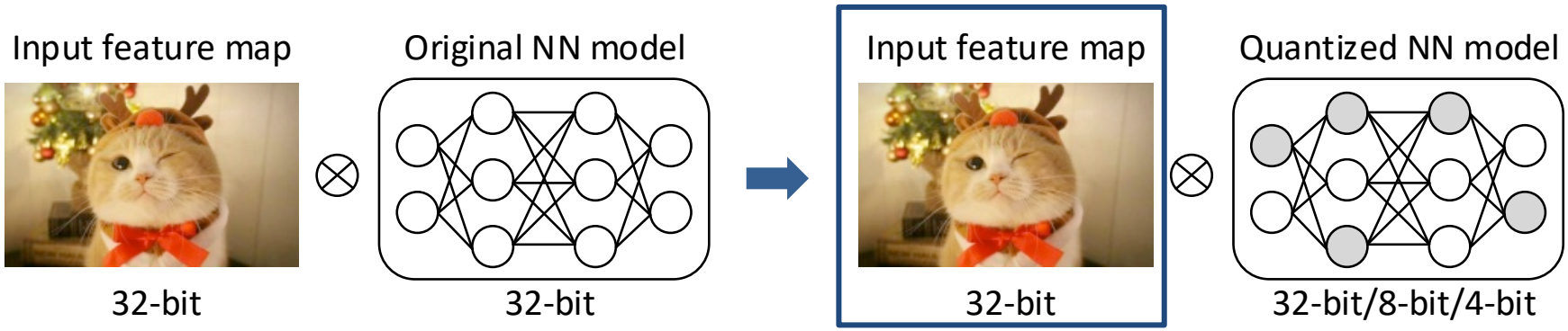
(b) Decomposing the 4-bit multiplication to four 2-bit multiplications.

(c) Mapping decomposed multiplications to BitBricks (BBs).

Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks (ISCA'18)

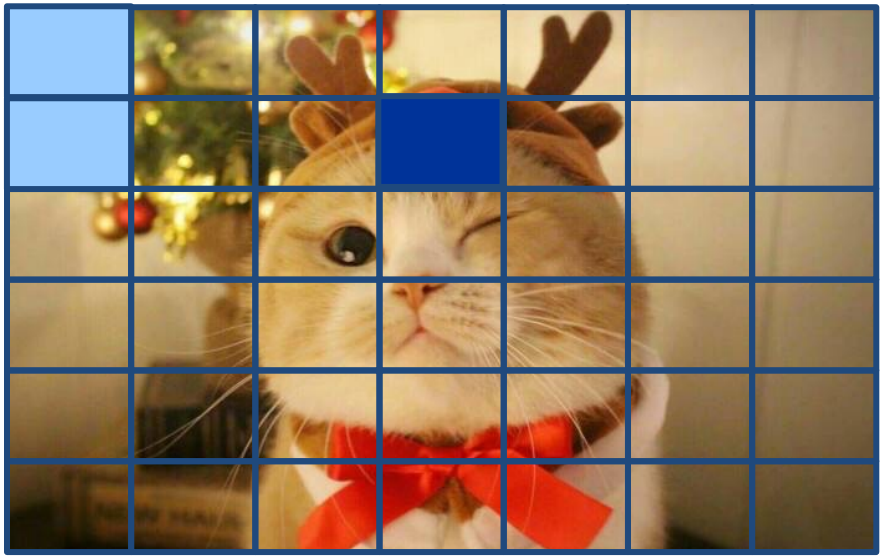
Mixed-precision Accelerator

- **Adaptive quantization: quantize weights using different bit-width**
  - *i.e., inter-layer quantization [cvpr' 19], intra-layer quantization [eccv' 18]*
- **Lack of exploration on input feature maps**



Quantization for Feature Map

- **Dynamic regions-based quantization**
  - *Insensitive region with low-precision convolution*
  - *Sensitive region with high-precision convolution*



Input feature map

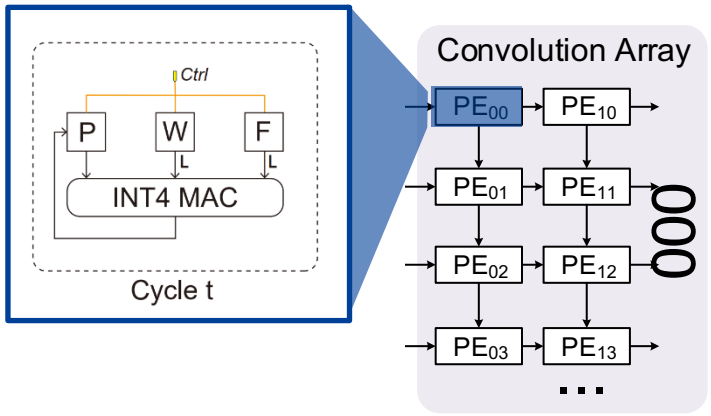
Low-precision **High-precision**

DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration (ISCA'20)

Quantization for Feature Map

How to support precision switching?

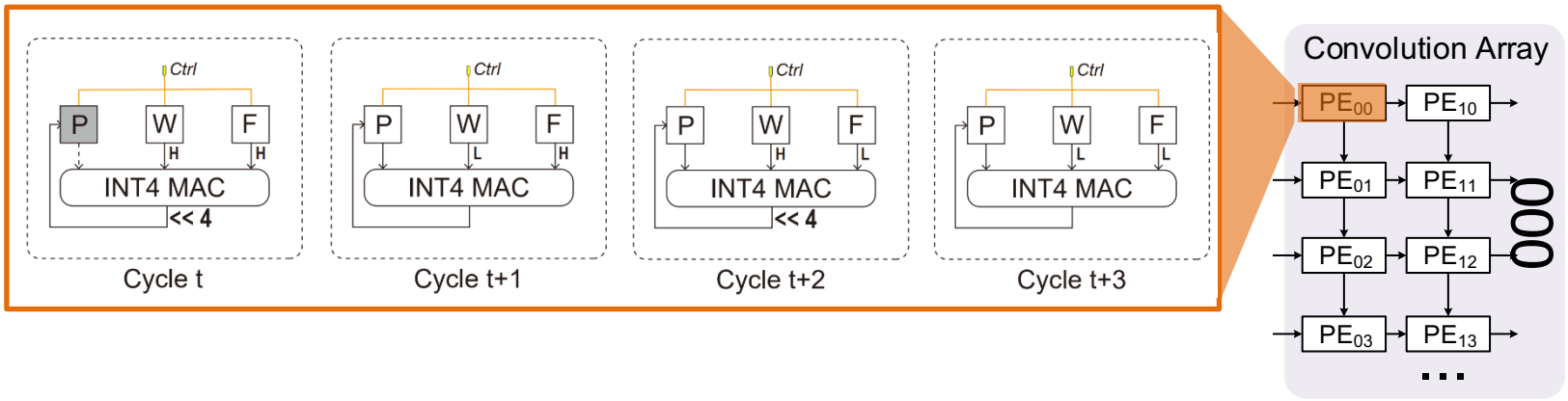
- **Multi-precision PE**
  - *Keep the 8-bit weight values in case of sensitive values*
  - *Support 4-bit MAC with one cycle latency*



Quantization for Feature Map

How to support precision switching?

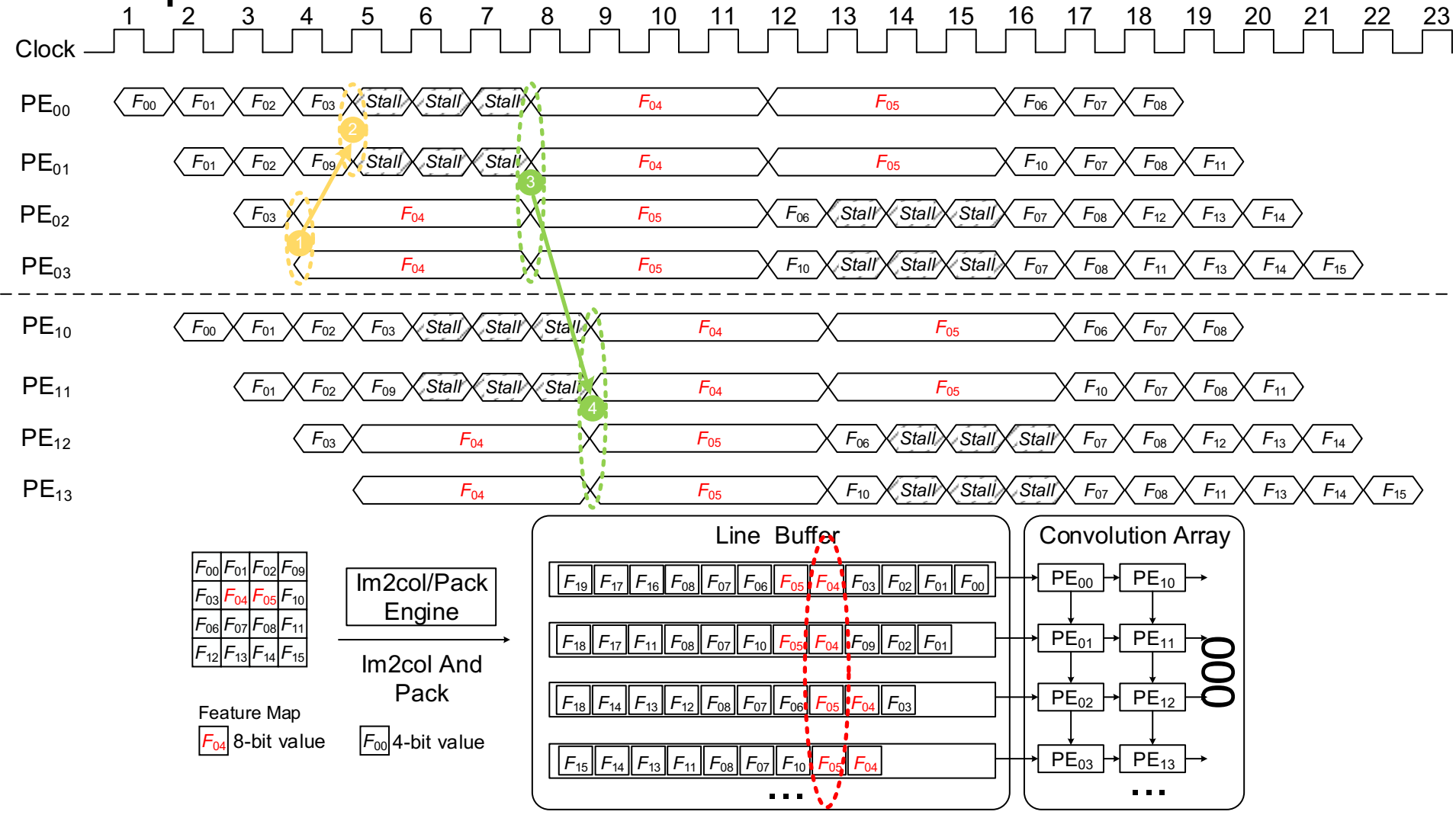
- **Multi-precision PE**
  - *Keep the 8-bit weight values in case of sensitive values*
  - *Support 4-bit MAC with one cycle latency*
  - *Support 8-bit MAC with four cycle latency*



DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration (ISCA'20)

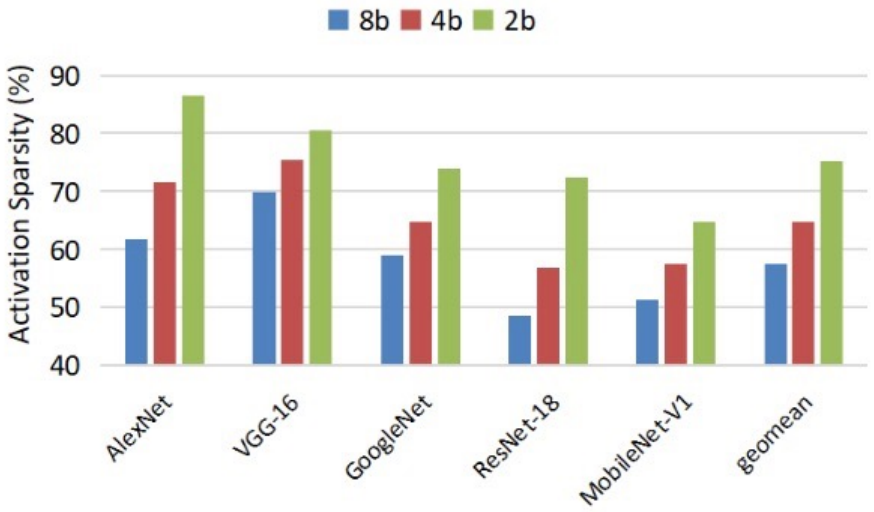
Quantization for Feature Map

- **Example**

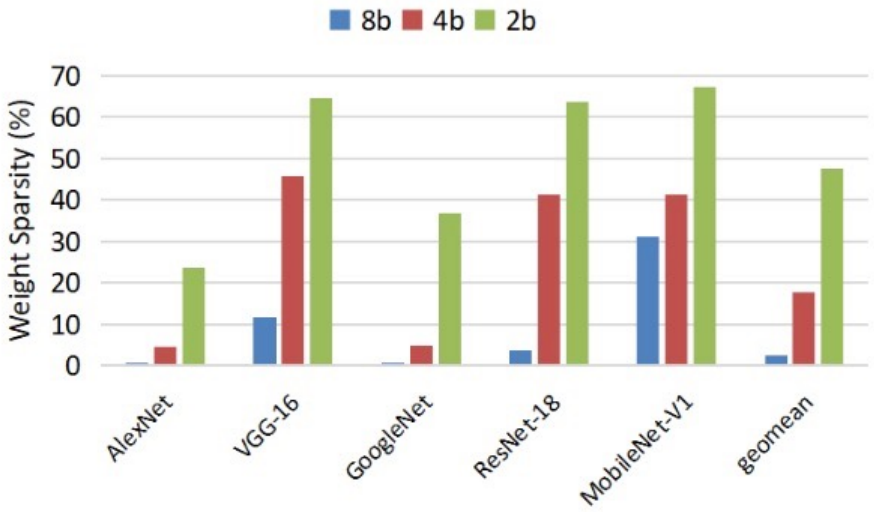


Pruning+ Quantization

- **DNN usually contains both sparsity and low-precision data**



(a) Activation Sparsity

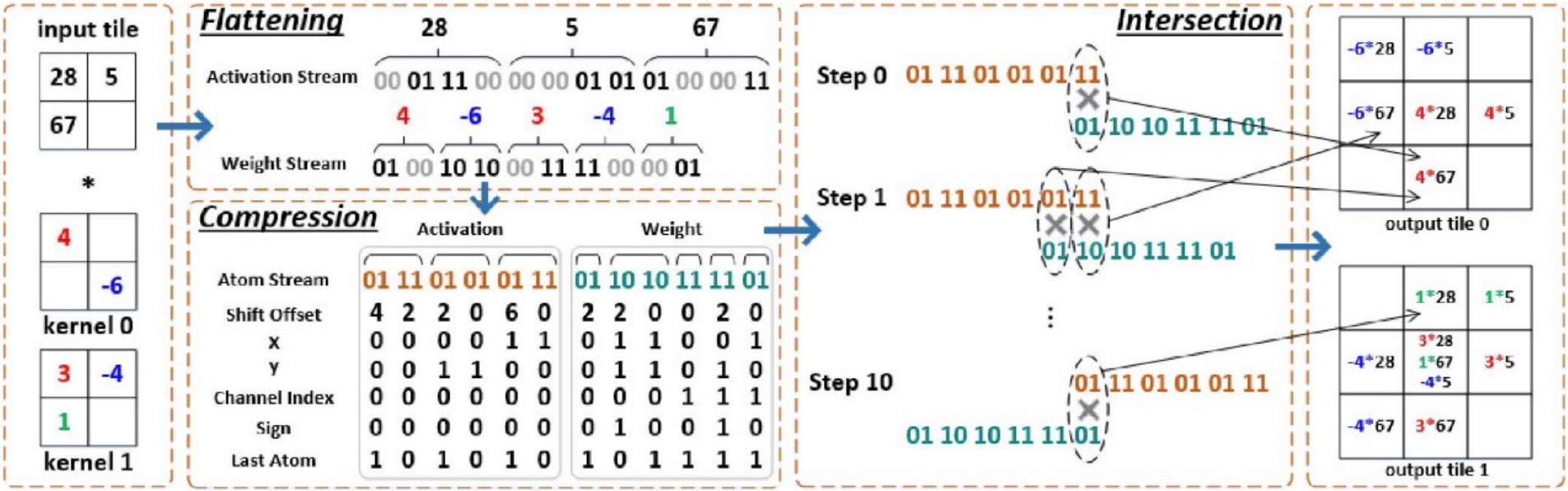


(b) Weight Sparsity

Ristretto: An Atomized Processing Architecture for Sparsity-Condensed Stream Flow in CNN (MICRO'22)

Pruning+ Quantization

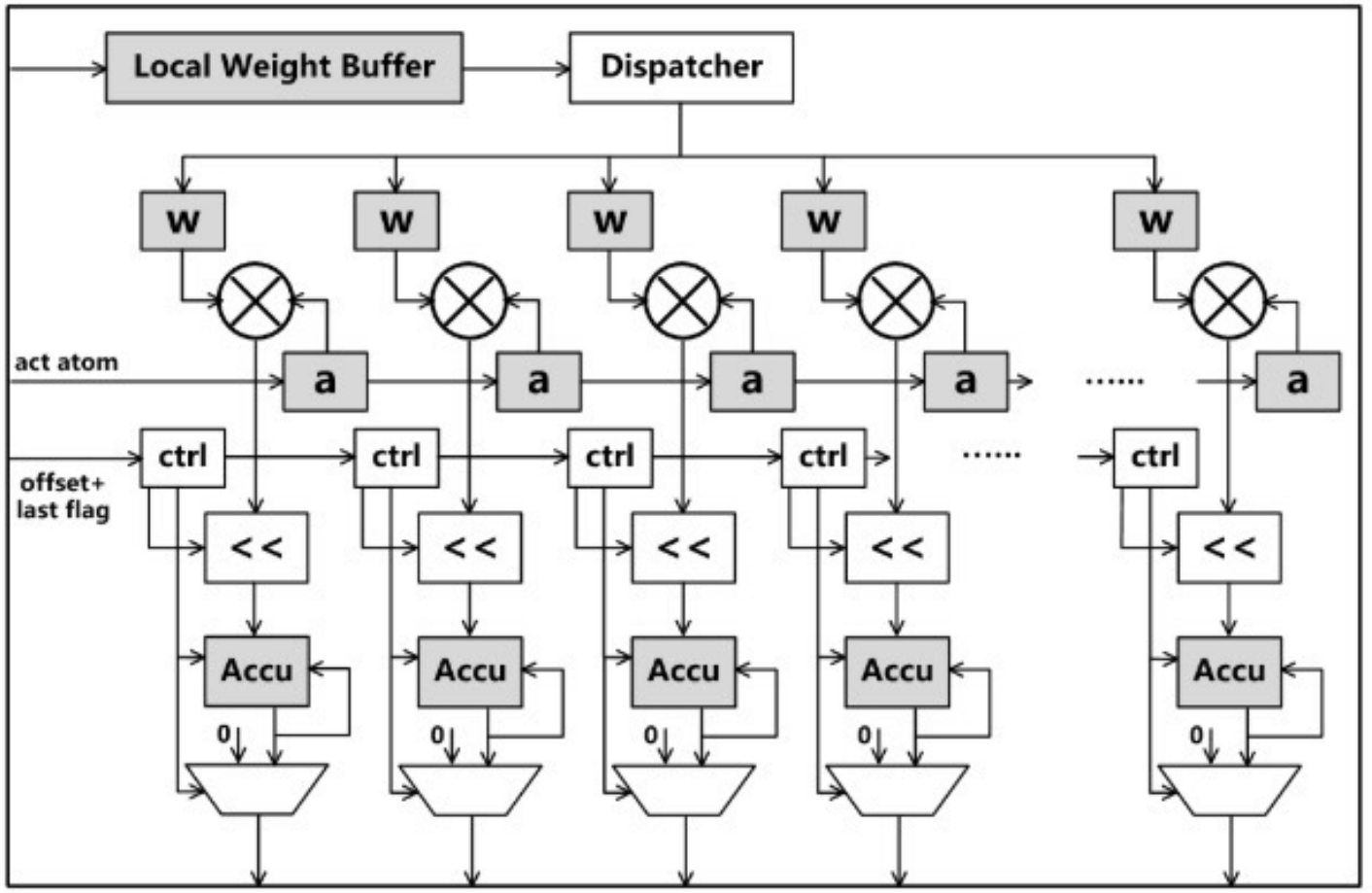
- **Atomize 2-bit computation**



Ristretto: An Atomized Processing Architecture for Sparsity-Condensed Stream Flow in CNN (MICRO'22)

Pruning+ Quantization

- **Basic computation component**



Ristretto: An Atomized Processing Architecture for Sparsity-Condensed Stream Flow in CNN (MICRO'22)