2022 Fall @ POSTECH

EECE695D: Efficient ML Systems

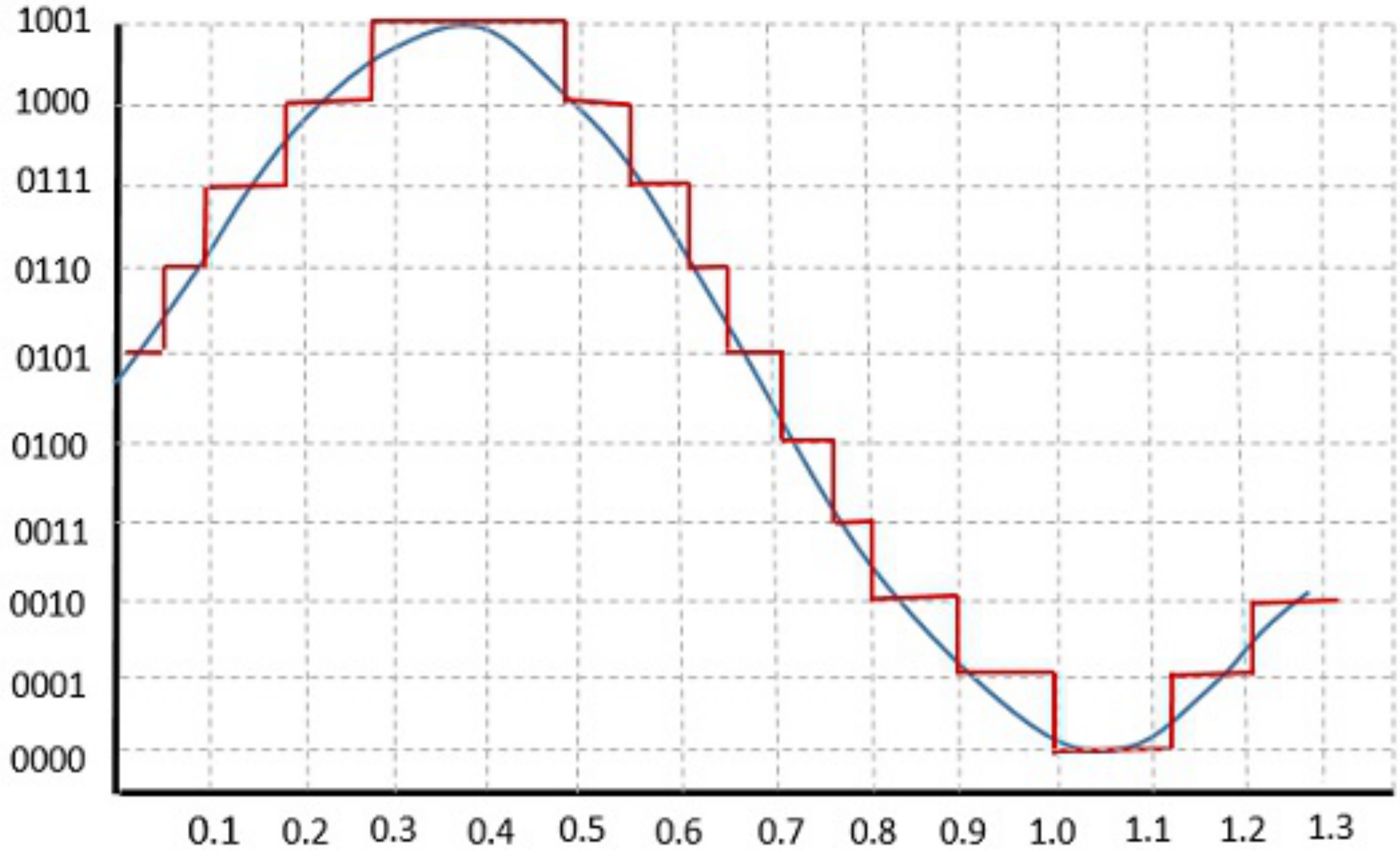
Quantization

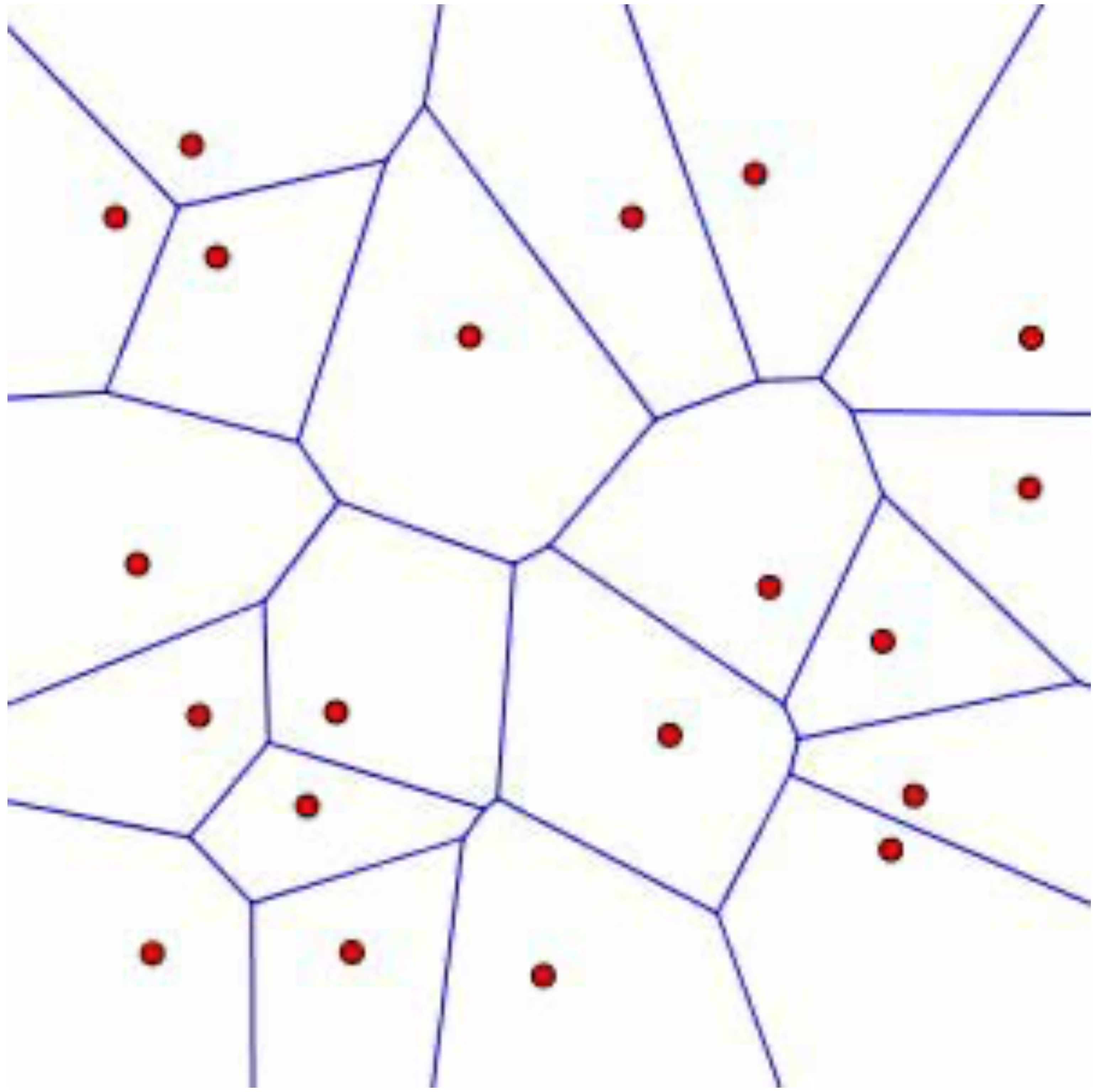
(Note: Many figures from Song Han’s slides)

Quantization

Roughly speaking, quantization is a mapping:

- From an input that belongs to a large set (e.g., ℝ, 32-bit floating point, …)
- To an output that belongs to a smaller, discrete set (e.g., FP32, 8-bit integer, …)

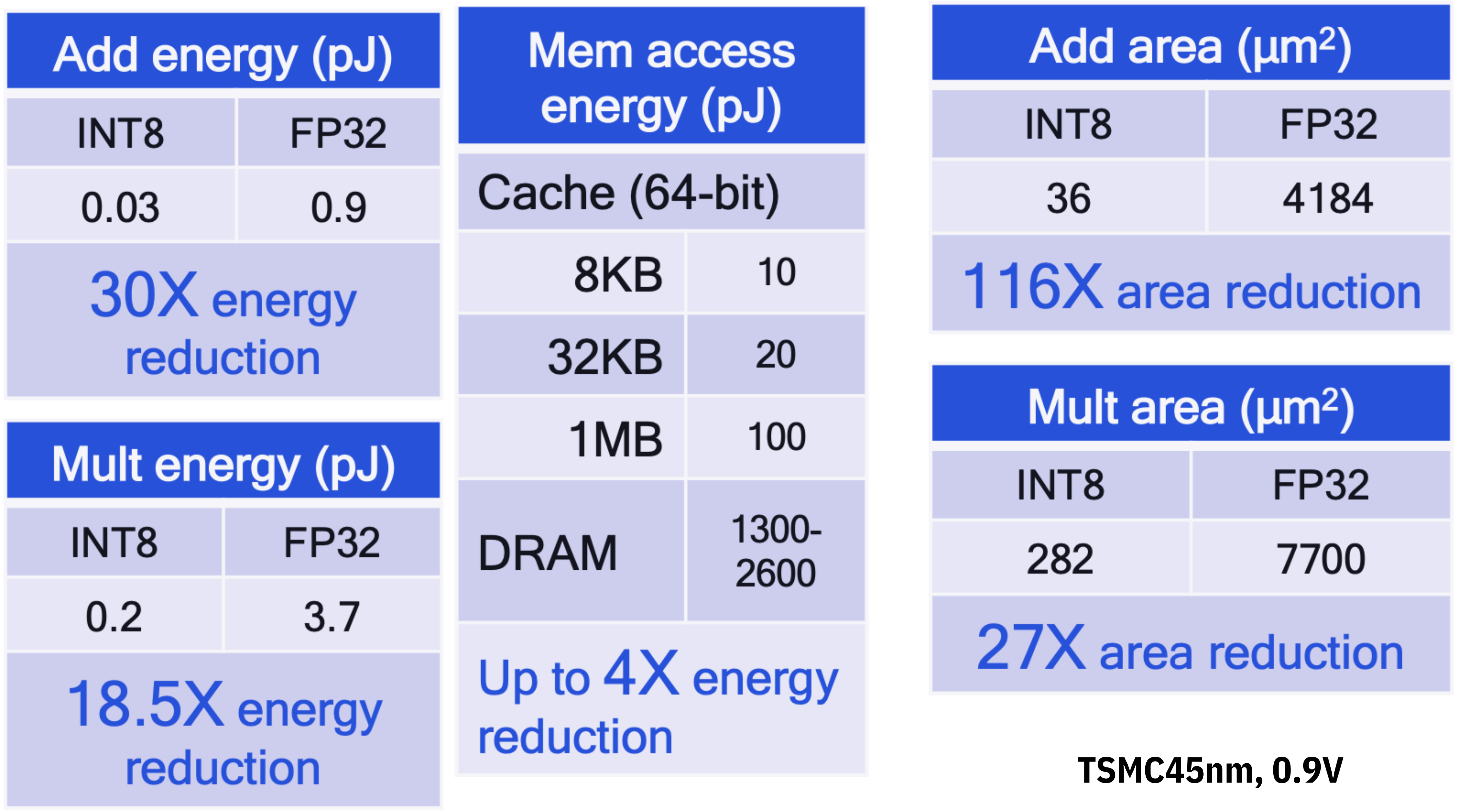




Quantization in Deep Learning

Typically a scalar quantization, dropping the bitwidth of the weights/activations (e.g., 32 bits -> 16/8/4 bits). Enjoys many benefits, including:

- Multiplying low-precision weights with low-precision activation requires less compute
- Loading low-precision weights/activations requires less memory bandwidth usage
- Storing quantized data/models requires less storage space
- Low bit processing typically requires less silicon space!



Add energy (pJ)	
INT8	FP32
0.03	0.9
30X energy reduction	

Mult energy (pJ)	
INT8	FP32
0.2	3.7
18.5X energy reduction	

Mem access energy (pJ)	
Cache (64-bit)	
8KB	10
32KB	20
1MB	100
DRAM	1300-2600
Up to 4X energy reduction	

Add area (μm^2)	
INT8	FP32
36	4184
116X area reduction	

Mult area (μm^2)	
INT8	FP32
282	7700
27X area reduction	

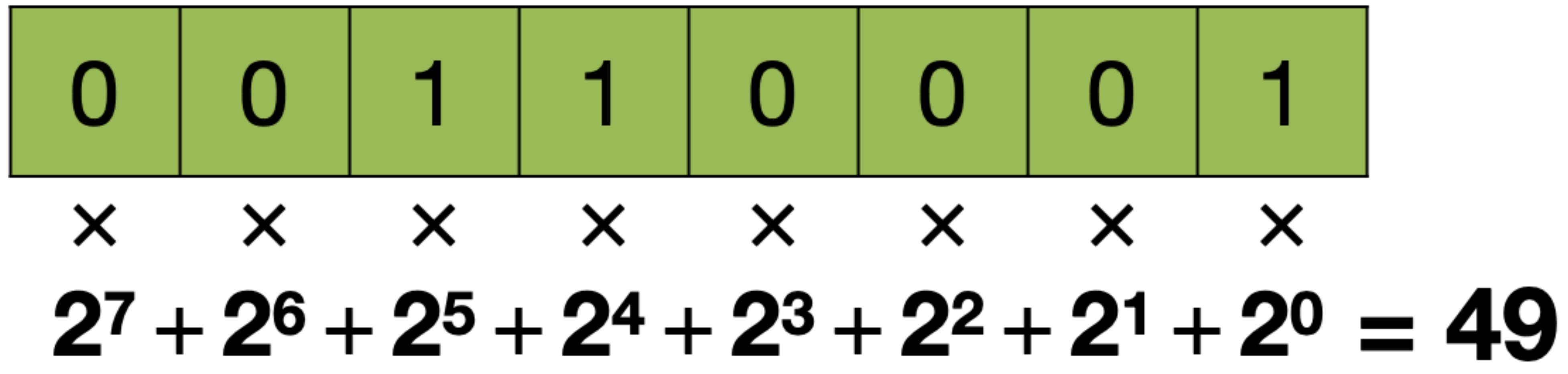
TSMC45nm, 0.9V

Horowitz, *"Computing's energy problem (and what we can do about it)," ISSCC 2014*
Fournakis *"A Practical Guide to NN Quantization" TinyML Talk 2021*

Recap: Data numerics

Integers. No fractions, just integers.

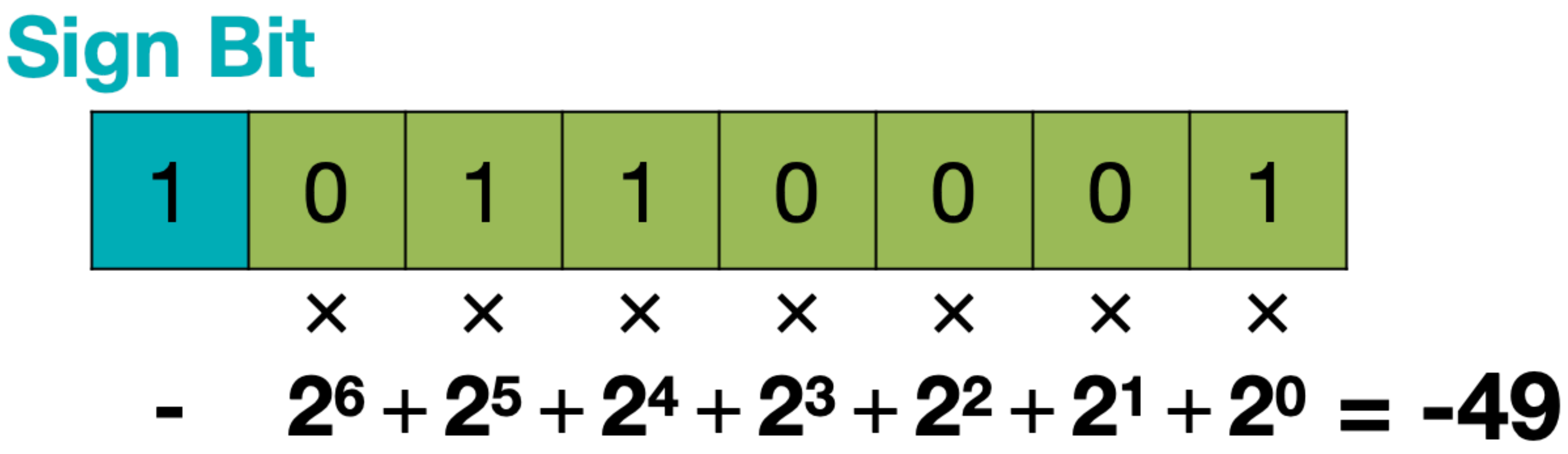
- *Unsigned Integer.* Can represent $\{0, 1, \dots, 2^n - 1\}$
- *Signed Integer* *(ver 1. Sign-magnitude).*
    - Can represent $\{-2^{n-1} - 1, \dots, 2^{n-1} - 1\}$
    - Both $000\cdots00$ and $100\cdots00$ represents 0
- *Signed Integer* *(ver 2. 2's complement).*
    - Can represent $\{-2^{n-1}, \dots, 2^{n-1} - 1\}$
    - $100\cdots00$ now represents -2^{n-1} instead of 0



0	0	1	1	0	0	0	1

× × × × × × × ×

$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 49$

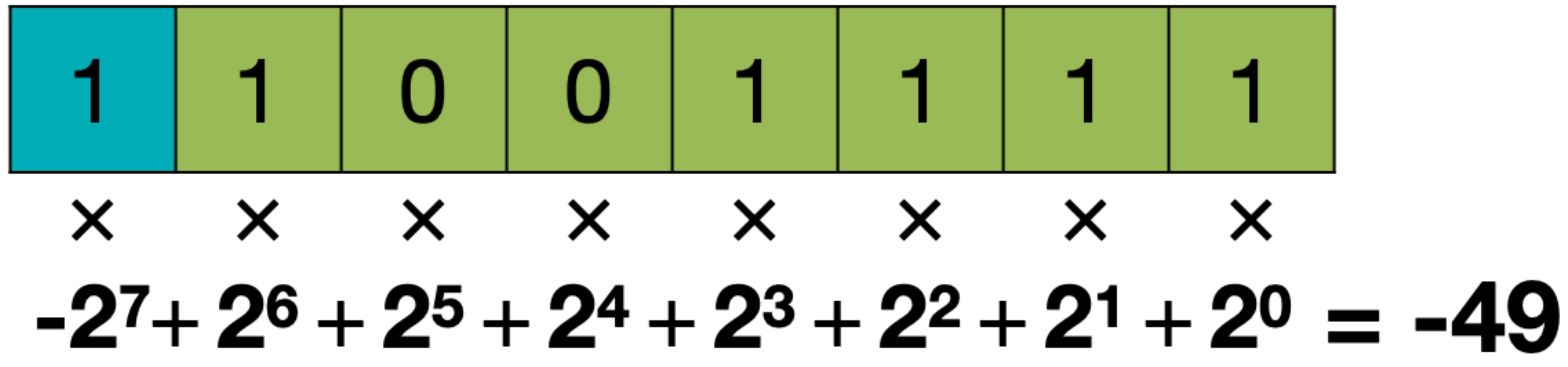


Sign Bit

1	0	1	1	0	0	0	1

× × × × × × ×

$- \quad 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -49$



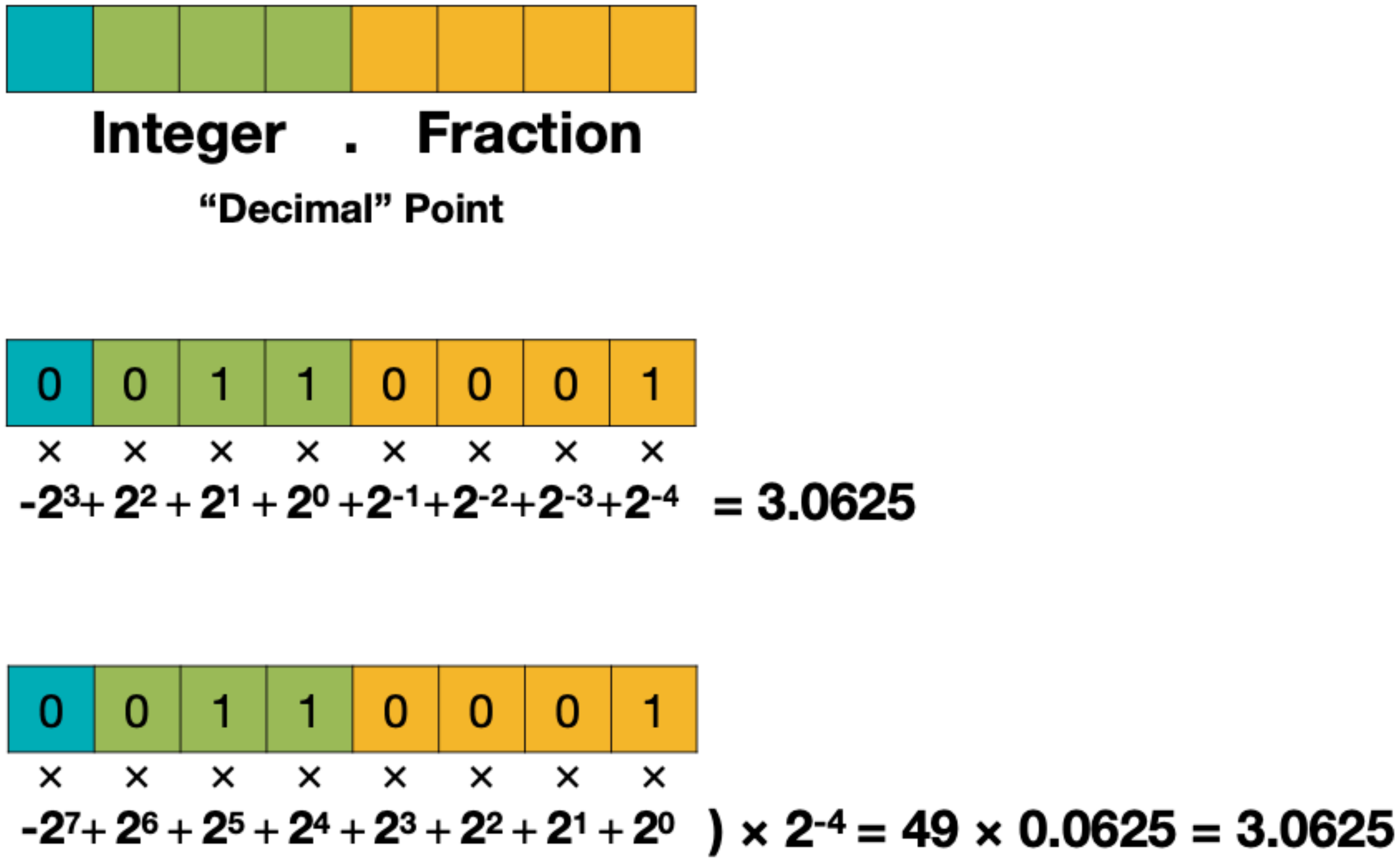
1	1	0	0	1	1	1	1

× × × × × × × ×

$-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -49$

Fixed-Point. Integers, but with fractions.

- Bits after the virtual decimal point represent fractions
- Mostly used under very specific circumstances (e.g., super low-cost microprocessors)
  Used in very early works (e.g., Vanhoucke et al., 2011; Hwang & Sung, 2014)



0	0	1	1	0	0	0	1
×	×	×	×	×	×	×	×

$$-2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 3.0625$$

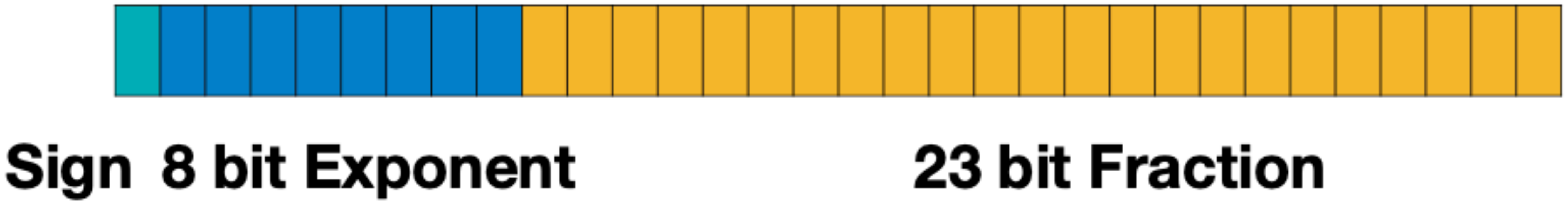
0	0	1	1	0	0	0	1
×	×	×	×	×	×	×	×

$$(-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0) \times 2^{-4} = 49 \times 0.0625 = 3.0625$$

Vanhoucke et al., “Improving the speed of neural networks on CPUs,” NeurIPS workshop 2011
Hwang & Sung, “Fixed-point feedforward DNN design using weights +1, 0, and -1,” IEEE SiPS workshop 2014.

Floating-Point. Fixed-point, but the scaling factor is not fixed

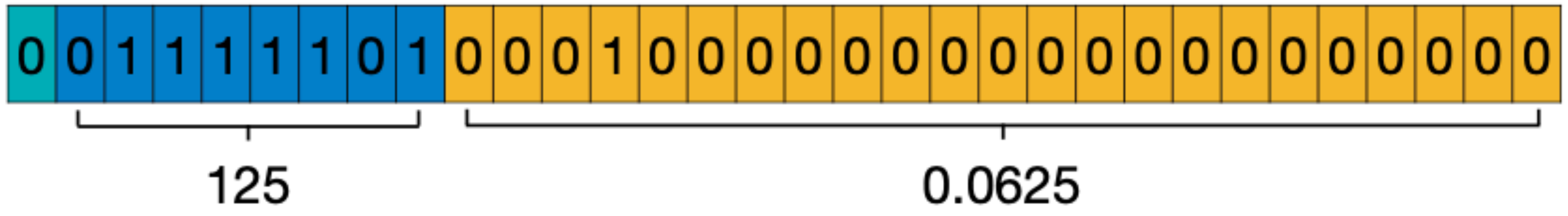
- Consists of a sign bit, exponent bits, and fraction bits (a.k.a. mantissa)
- Exponent = Dynamic range (important for accumulation), Fraction = Precision

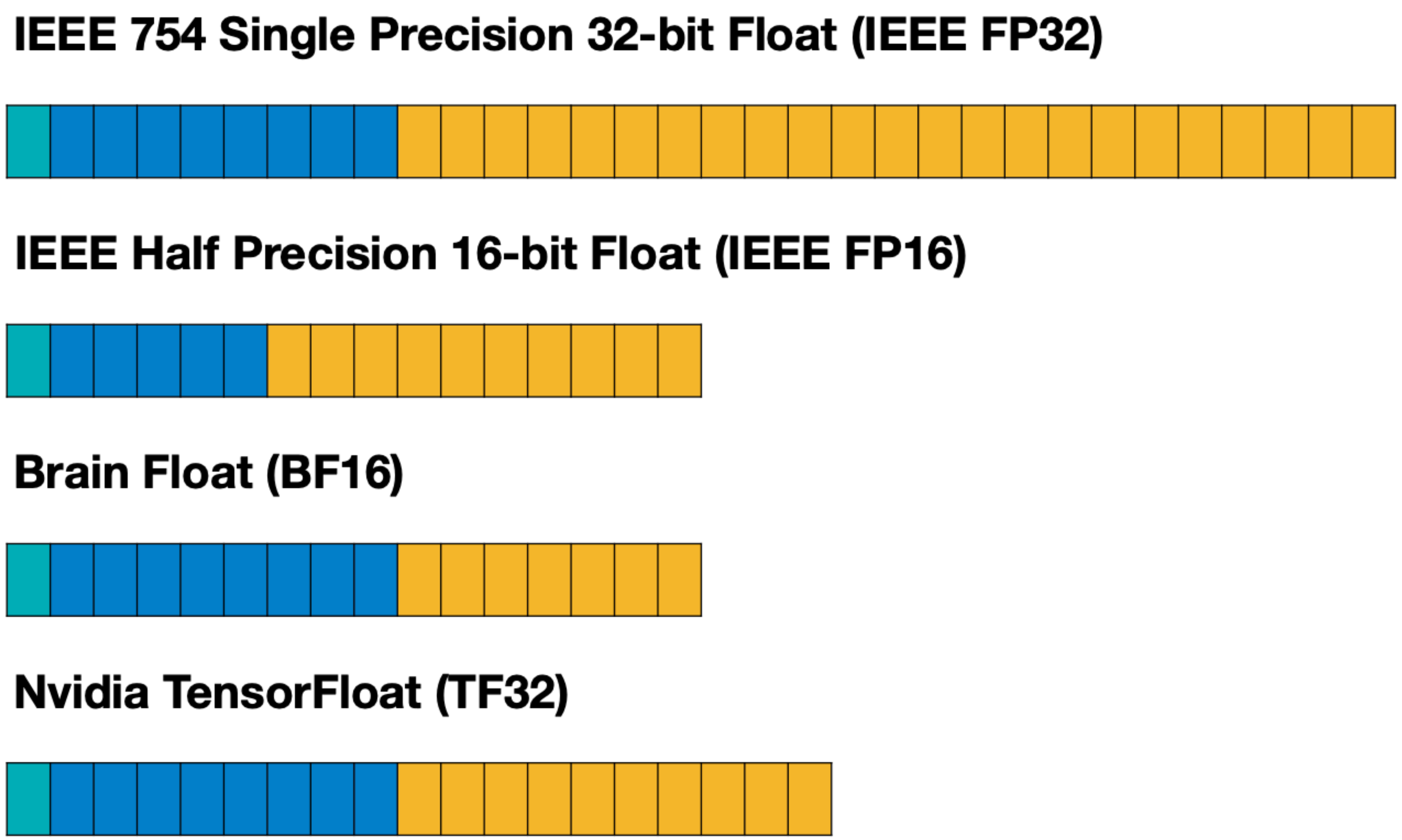


$$(-1)^{\text{sign}} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$$ ← Exponent Bias = 127 = $2^{8-1}-1$

(significant / mantissa)

$$0.265625 = 1.0625 \times 2^{-2} = (1 + \underline{0.0625}) \times 2^{\underline{125}-127}$$




IEEE 754 Single Precision 32-bit Float (IEEE FP32)
IEEE Half Precision 16-bit Float (IEEE FP16)
Brain Float (BF16)
Nvidia TensorFloat (TF32)

Exponent (bits)	Fraction (bits)	Total (bits)
8	23	32
5	10	16
8	7	16
8	10	19

BF16. Can replace/be-combined-with FP32 (identical underflow, overflow, NaN, ...)

Typical MAC: Multiply in bfloat16, Accumulate in FP32

Usually no “loss scaling” required

Smaller mantissa than FP16 -> smaller size needed in silicon

TF32. Exponent of FP32, Mantissa of FP16.

Used in NVIDIA Ampere.

More fraction -> better precision -> better performance.

Naturally, slower than BF/FP16

FP32	TF32	FP16 / BF16
1x	8x	16x

Table 1. Relative throughput of A100 GPU math.

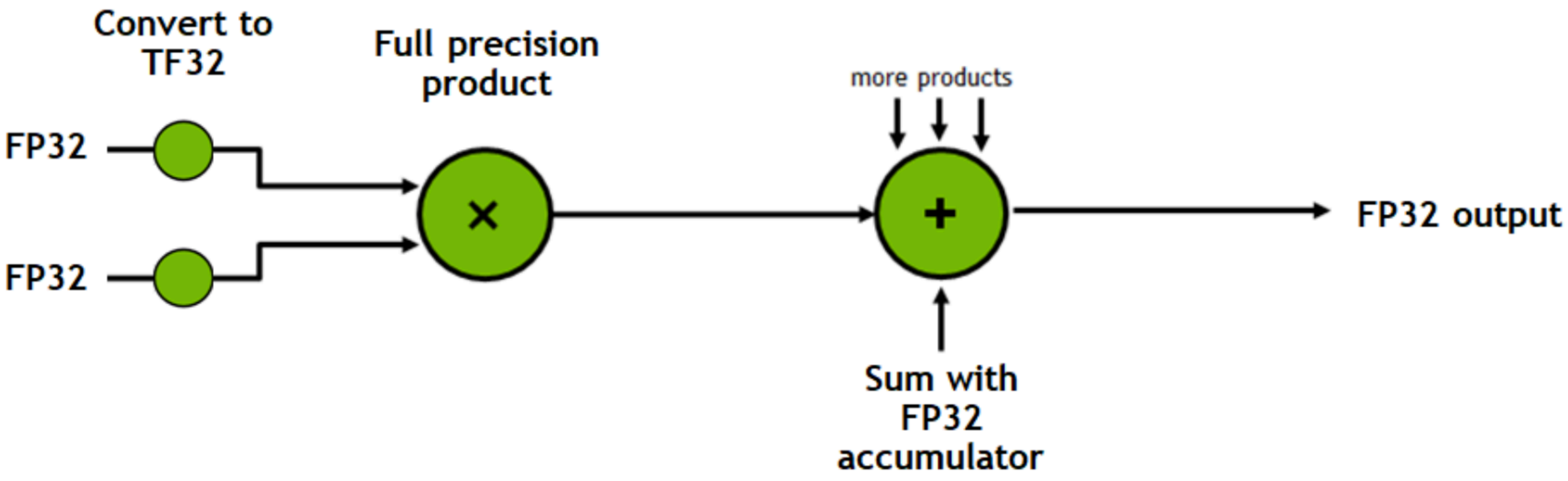


Figure 1. Ampere A100 Tensor Core operation.

A100 speedup over V100 FP32
TF32
FP16
Speedup
0
2
4
6
8
10
12
14
16
BERT FT 6x 15x
BERT PT 6x 11x
XL Large 5x 8x
Transformer 8x 12x
GNMT 5x 8x
DLRM 6x 9x
NCF 6x 7x
JASPER 4x 7x
WaveGlow 2x 5x
U-Net 3x 8x
ResNeXt101 4x 11x
ResNet50 3x 6x
VGG16 2x 4x
Inception V3 2x 3x
SSD 2x 4x
Mask R-CNN 3x 4x
Language Modeling
Translation
Recommendation
Speech
Image Classification
Detection

NVIDIA tech blog

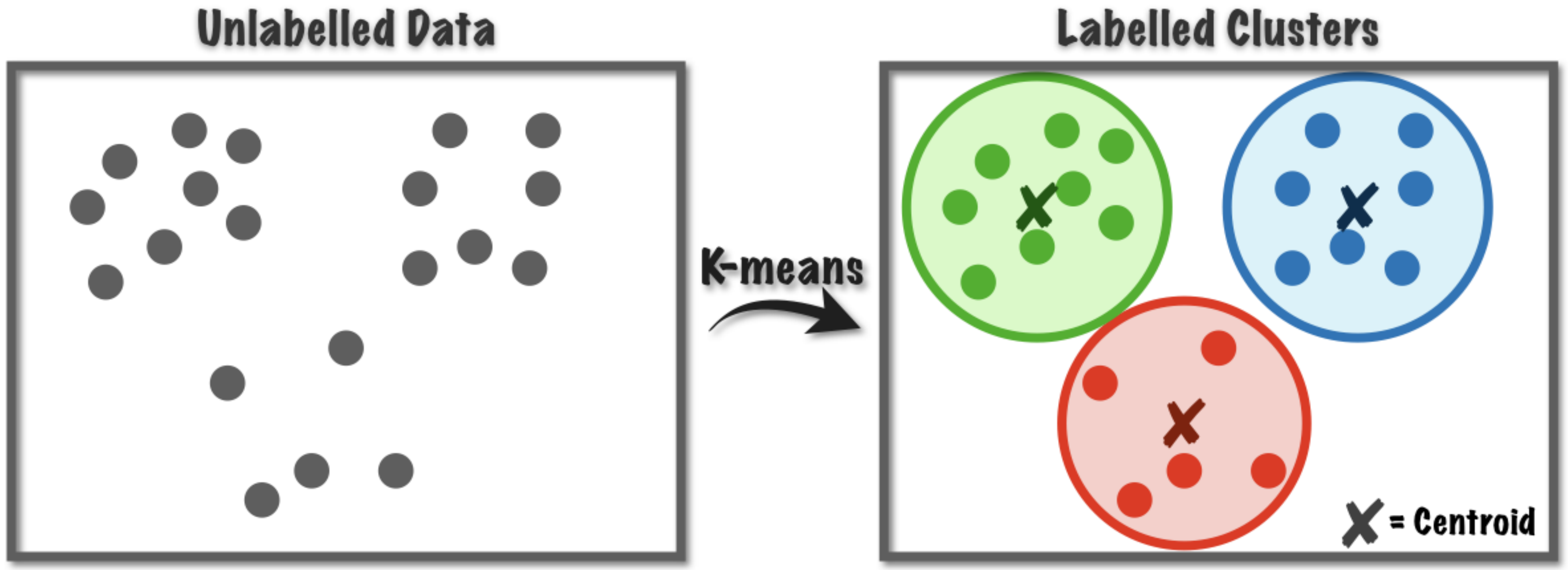
K-means based quantization

Around 2016, fancy methods based on K-means gained popularity (e.g., Han et al. 2016)

(1D) K-Means. Given $x_1, \ldots, x_n \in \mathbb{R}$, find $c_1, \ldots, c_k \in \mathbb{R}$ that minimizes

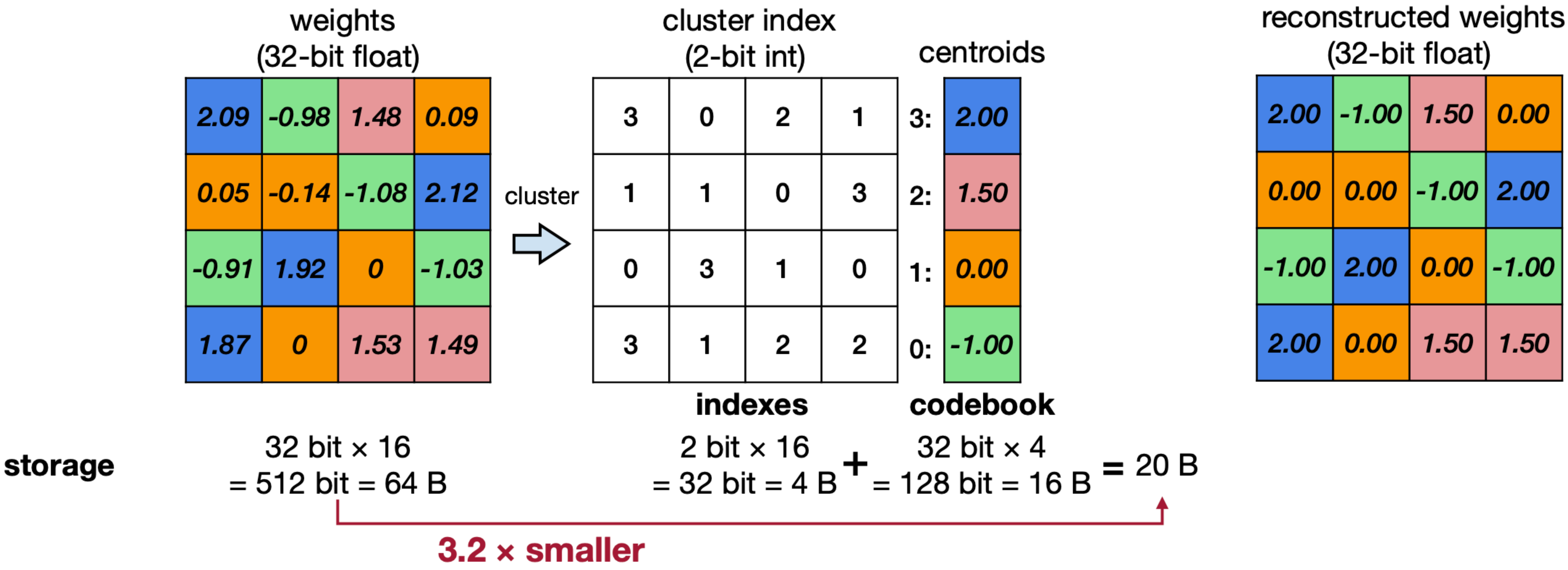
$$\frac{1}{n}\sum_{i=1}^{n} \min_{j\in[k]} (x_i - c_k)^2$$

Famous methods to solve this include: Lloyd's algorithm, k-means++, ...



Han et al., "Deep Compression: Compressing DNNs with pruning, trained quantization, and Huffman coding," ICLR 2016
https://towardsdatascience.com/k-means-a-complete-introduction-1702af9cd8c

Idea. Do the same thing for the weight matrix of a neural net
Need to store both index and codebook.



weights (32-bit float)

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

cluster index (2-bit int)

3	0	2	1
1	1	0	3
0	3	1	0
3	1	2	2

indexes

centroids

3:	2.00
2:	1.50
1:	0.00
0:	-1.00

codebook

reconstructed weights (32-bit float)

2.00	-1.00	1.50	0.00
0.00	0.00	-1.00	2.00
-1.00	2.00	0.00	-1.00
2.00	0.00	1.50	1.50

storage

32 bit × 16
= 512 bit = 64 B

2 bit × 16
= 32 bit = 4 B

+

32 bit × 4
= 128 bit = 16 B

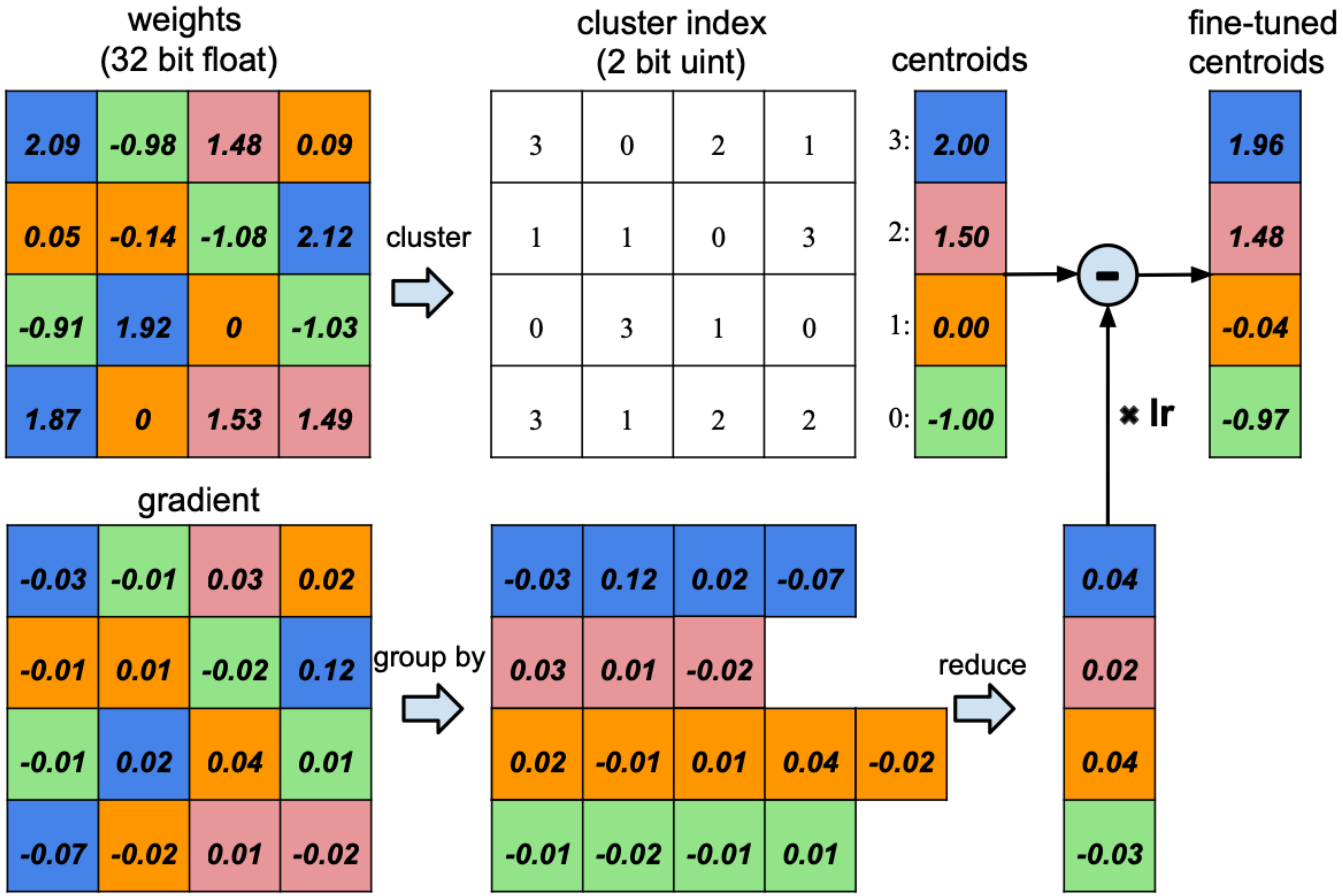
= 20 B

3.2 × smaller

Han et al., "Deep Compression: Compressing DNNs with pruning, trained quantization, and Huffman coding," ICLR 2016

Codebook training. Gradient updates can be done as in convolutional neural networks.

Note: Not for training neural net from scratch; only for compressing model itself!



weights (32 bit float)

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

cluster index (2 bit uint)

3	0	2	1
1	1	0	3
0	3	1	0
3	1	2	2

	centroids	fine-tuned centroids
3:	2.00	1.96
2:	1.50	1.48
1:	0.00	-0.04
0:	-1.00	-0.97

gradient

-0.03	-0.01	0.03	0.02
-0.01	0.01	-0.02	0.12
-0.01	0.02	0.04	0.01
-0.07	-0.02	0.01	-0.02

group by

-0.03	0.12	0.02	-0.07	
0.03	0.01	-0.02		
0.02	-0.01	0.01	0.04	-0.02
-0.01	-0.02	-0.01	0.01	

reduce

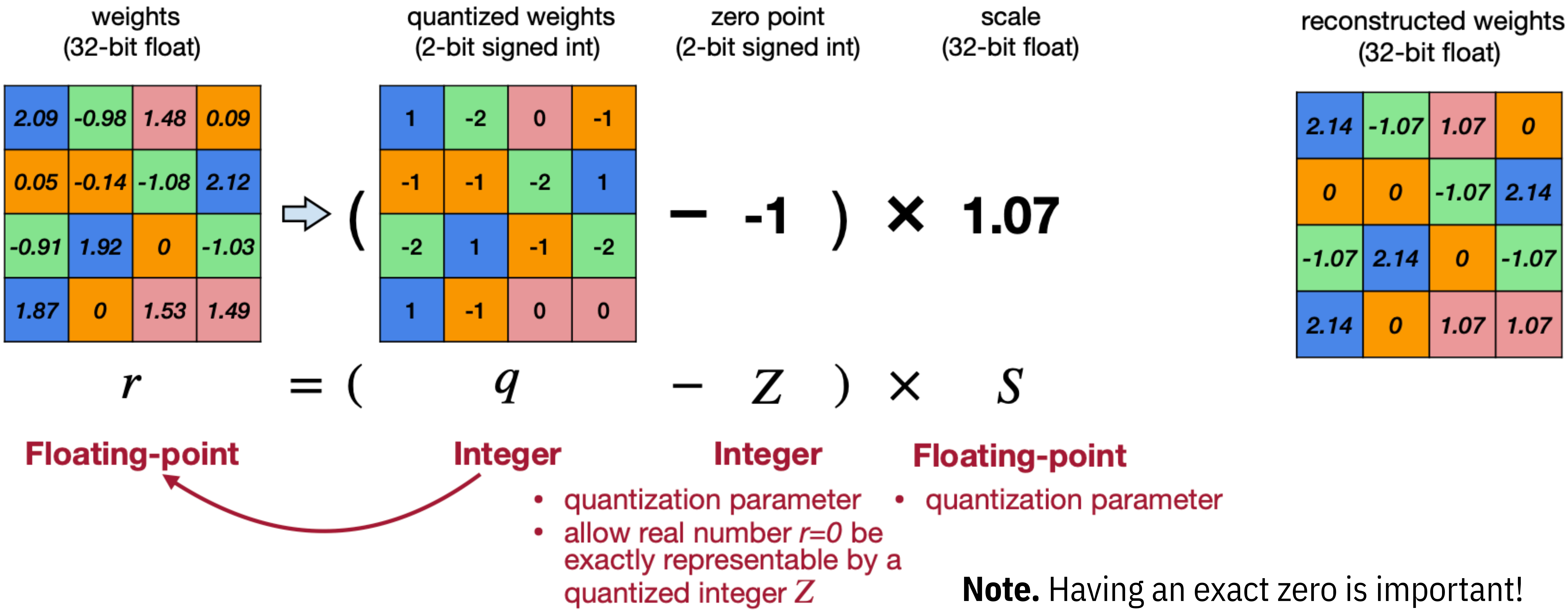
0.04
0.02
0.04
-0.03

Limitation. Reduces weight storage (INT index, FP codebook) but not compute/activation (uses FP arithmetics).
Usually a common drawback of “nonlinear” methods.

Han et al., “Deep Compression: Compressing DNNs with pruning, trained quantization, and Huffman coding,” ICLR 2016

Linear quantization

More mainstream nowadays; weight matrix $\mathbf{r}$ is represented as $\mathbf{r} = S(\mathbf{q} - Z)$ (with S, Z being scalar)
Typically introduces more quantization error than nonlinear (but benefits usually outweigh)



weights (32-bit float) | quantized weights (2-bit signed int) | zero point (2-bit signed int) | scale (32-bit float) | reconstructed weights (32-bit float)

$$\begin{bmatrix}2.09 & -0.98 & 1.48 & 0.09\\ 0.05 & -0.14 & -1.08 & 2.12\\ -0.91 & 1.92 & 0 & -1.03\\ 1.87 & 0 & 1.53 & 1.49\end{bmatrix} \Rightarrow \left(\begin{bmatrix}1 & -2 & 0 & -1\\ -1 & -1 & -2 & 1\\ -2 & 1 & -1 & -2\\ 1 & -1 & 0 & 0\end{bmatrix} - \text{-1}\right) \times 1.07 \qquad \begin{bmatrix}2.14 & -1.07 & 1.07 & 0\\ 0 & 0 & -1.07 & 2.14\\ -1.07 & 2.14 & 0 & -1.07\\ 2.14 & 0 & 1.07 & 1.07\end{bmatrix}$$

$$r = (q - Z) \times S$$

Floating-point | **Integer** | **Integer** | **Floating-point**

Integer:
- quantization parameter
- allow real number *r=0* be exactly representable by a quantized integer Z

Floating-point:
- quantization parameter

Note. Having an exact zero is important!
(we've seen naturally arising sparsity)

Quant vs. Dequant. The previous formula

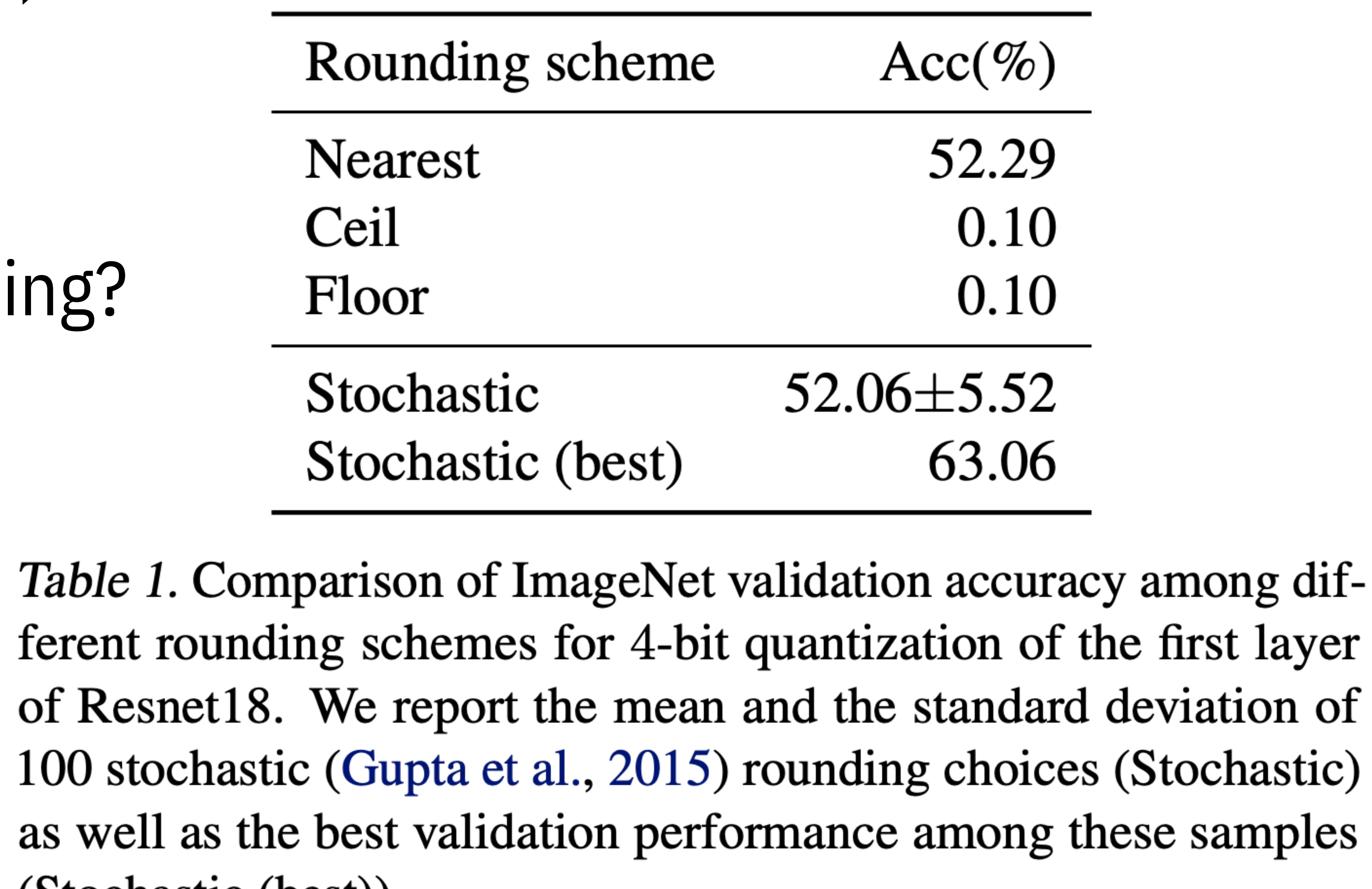
$$\mathbf{r} = S(\mathbf{q} - Z)$$

is actually what we call dequantization.

The formula for quantization can be written as:

$$\mathbf{q} = \mathsf{Clip}\left(\mathsf{Round}\left(\frac{\mathbf{r}}{S}\right) + Z \right)$$

Q. Does how-to-round matter? Can we improve by loss-oriented rounding?
A. Yes, see AdaRound paper by Nagel et al. (2020)



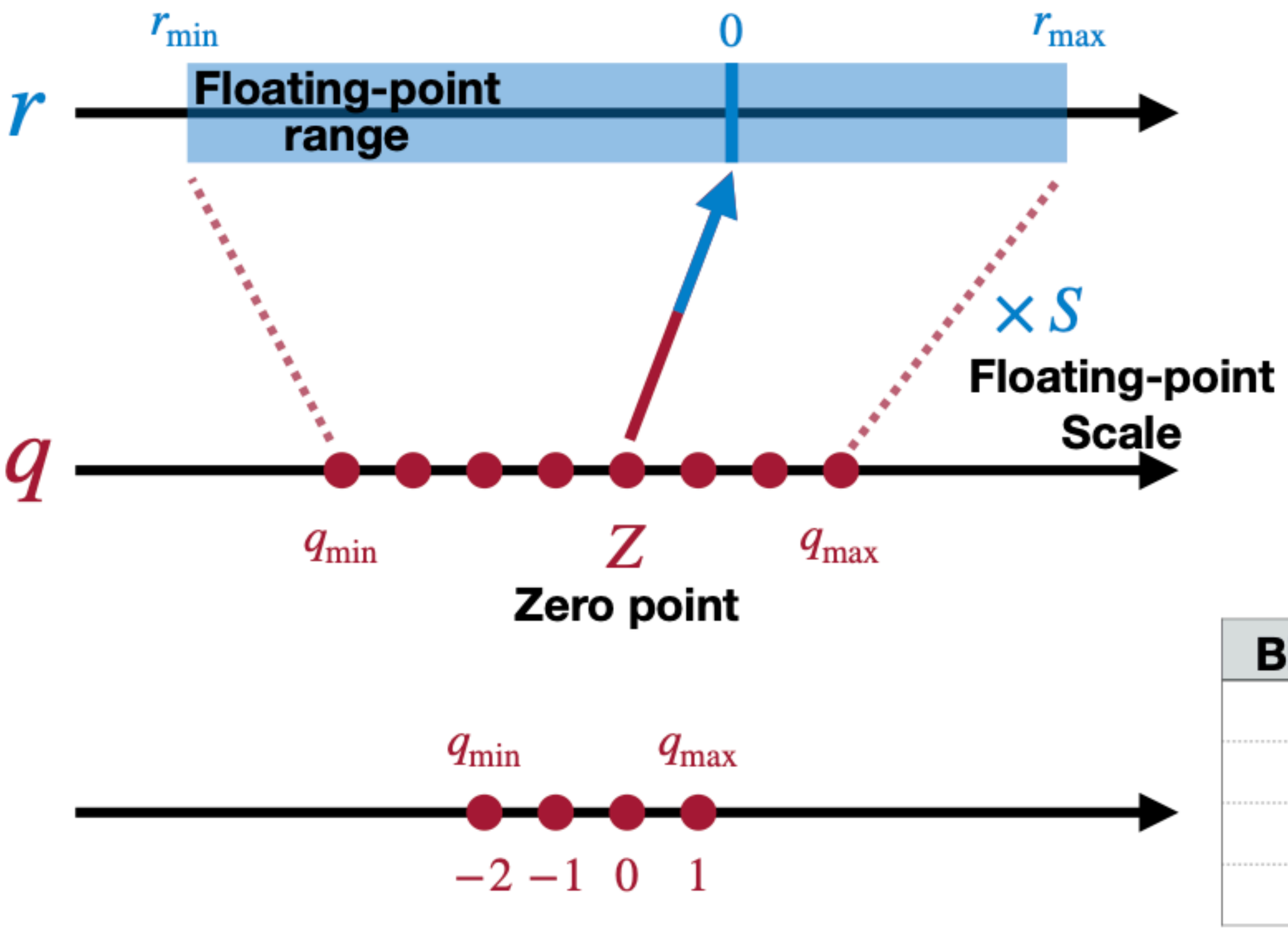
Rounding scheme	Acc(%)
Nearest	52.29
Ceil	0.10
Floor	0.10
Stochastic	52.06±5.52
Stochastic (best)	63.06

Table 1. Comparison of ImageNet validation accuracy among different rounding schemes for 4-bit quantization of the first layer of Resnet18. We report the mean and the standard deviation of 100 stochastic (Gupta et al., 2015) rounding choices (Stochastic) as well as the best validation performance among these samples (Stochastic (best)).

Nagel et al., "Adaptive Rounding for Post-Training Quantization," ICML 2020

Q. Given a matrix $\mathbf{r}$, How should we optimize the scale S and zero Z?

A. Typical method: Decide S to match max & min, then decide Z to be the nearest point.



2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$$

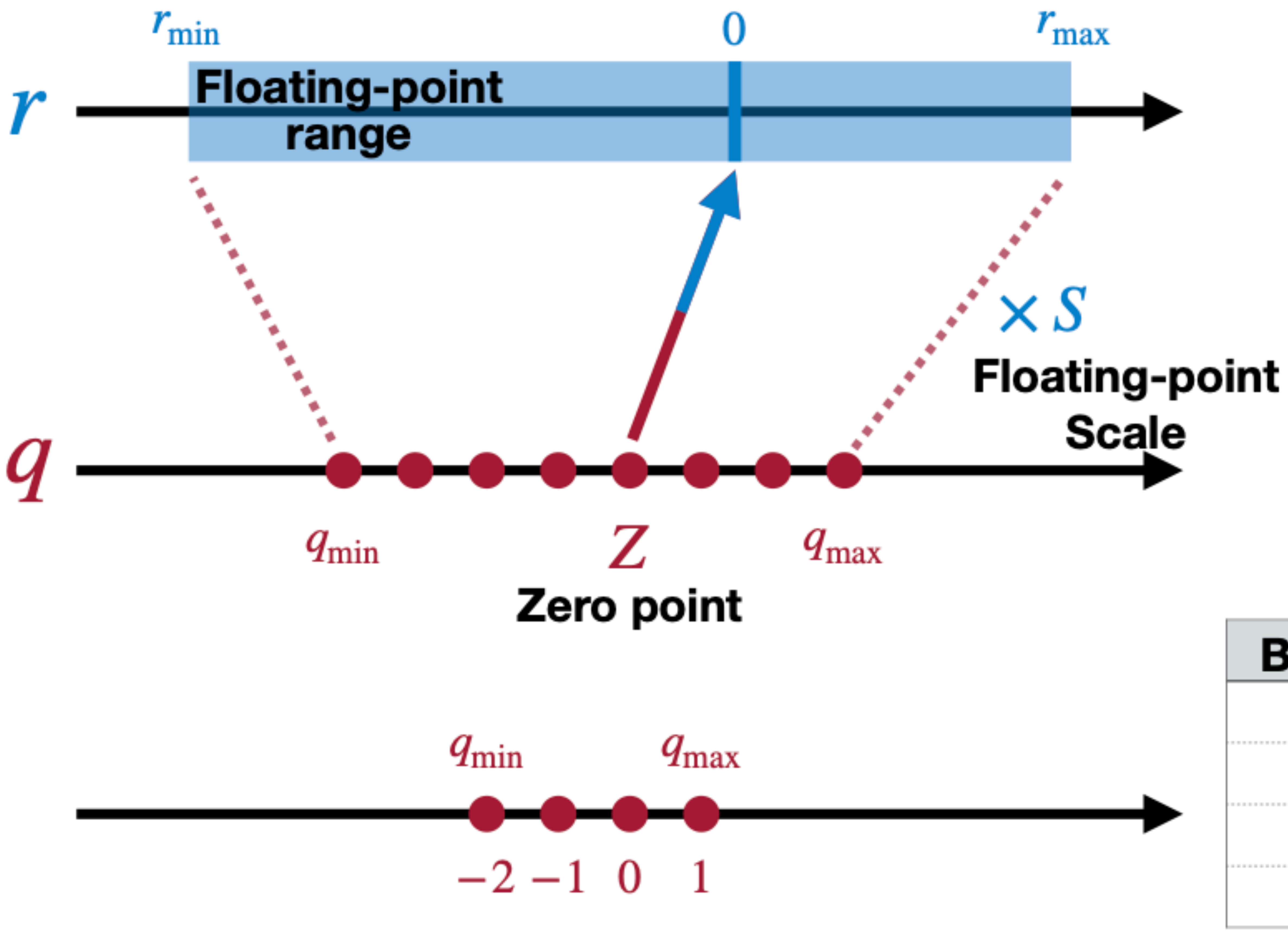
Binary	Decimal
01	1
00	0
11	-1
10	-2

$$= \frac{2.12 - (-1.08)}{1 - (-2)}$$

$$= 1.07$$

Q. Given a matrix $\mathbf{r}$, How should we optimize the scale S and zero Z?

A. Typical method: Decide S to match max & min, then decide Z to be the nearest point.



2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Binary	Decimal
01	1
00	0
11	-1
10	-2

$$Z = q_{\min} - \frac{r_{\min}}{S}$$

$$= \text{round}(-2 - \frac{-1.08}{1.07})$$

$$= -1$$

Dot products can now be done more efficiently.

Suppose that we are doing

$$Y = W^\top X, \qquad W, X \in \mathbb{R}^d, Y \in \mathbb{R}^1$$

where we have $W = S_W(\mathbf{q}_W - Z_W)$, $X = S_X(\mathbf{q}_X - Z_X)$, and $Y = S_Y(\mathbf{q}_Y - Z_Y)$.
(Here, let Z be a vectorized form $Z = z \cdot \mathbf{1}$)

In other words, we are doing

$$S_Y(\mathbf{q}_Y - Z_Y) = \left(S_W S_X\right) \cdot \left(\mathbf{q}_W\mathbf{q}_X - Z_X\mathbf{q}_W - Z_W\mathbf{q}_X + Z_W Z_X\right)$$

Assume that we are not dynamically adjusting S, Z. Then, $\mathbf{q}_Y$ can be written as:

$$\mathbf{q}_Y = \frac{S_W S_X}{S_Y}\left(\mathbf{q}_W^\top\mathbf{q}_X - Z_W^\top\mathbf{q}_X - Z_X^\top\mathbf{q}_W + Z_W^\top Z_X\right) + Z_Y$$

Much easier when
$Z_W = 0$ (symmetric quantization!)

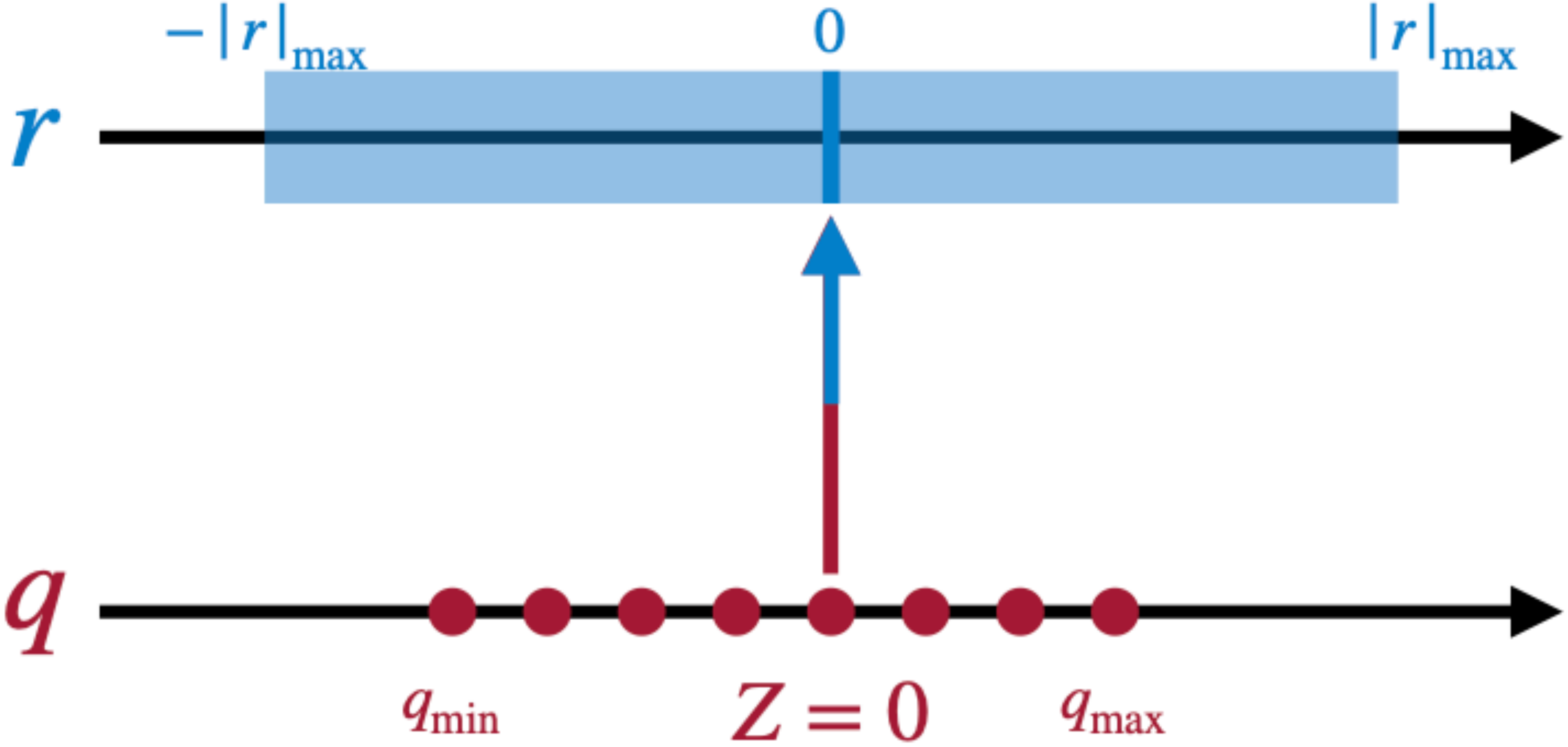
Should be rescaled to
a low-bit integer
value empirically in $(0,1)$

Can be pre-computed
and folded into bias

$$\mathbf{q}_Y = \boxed{\frac{S_W S_X}{S_Y}} \boxed{\left(\mathbf{q}_W^\top \mathbf{q}_X - \boxed{Z_W^\top \mathbf{q}_X} - \boxed{Z_X^\top \mathbf{q}_W + Z_W^\top Z_X}\right)} + Z_Y$$

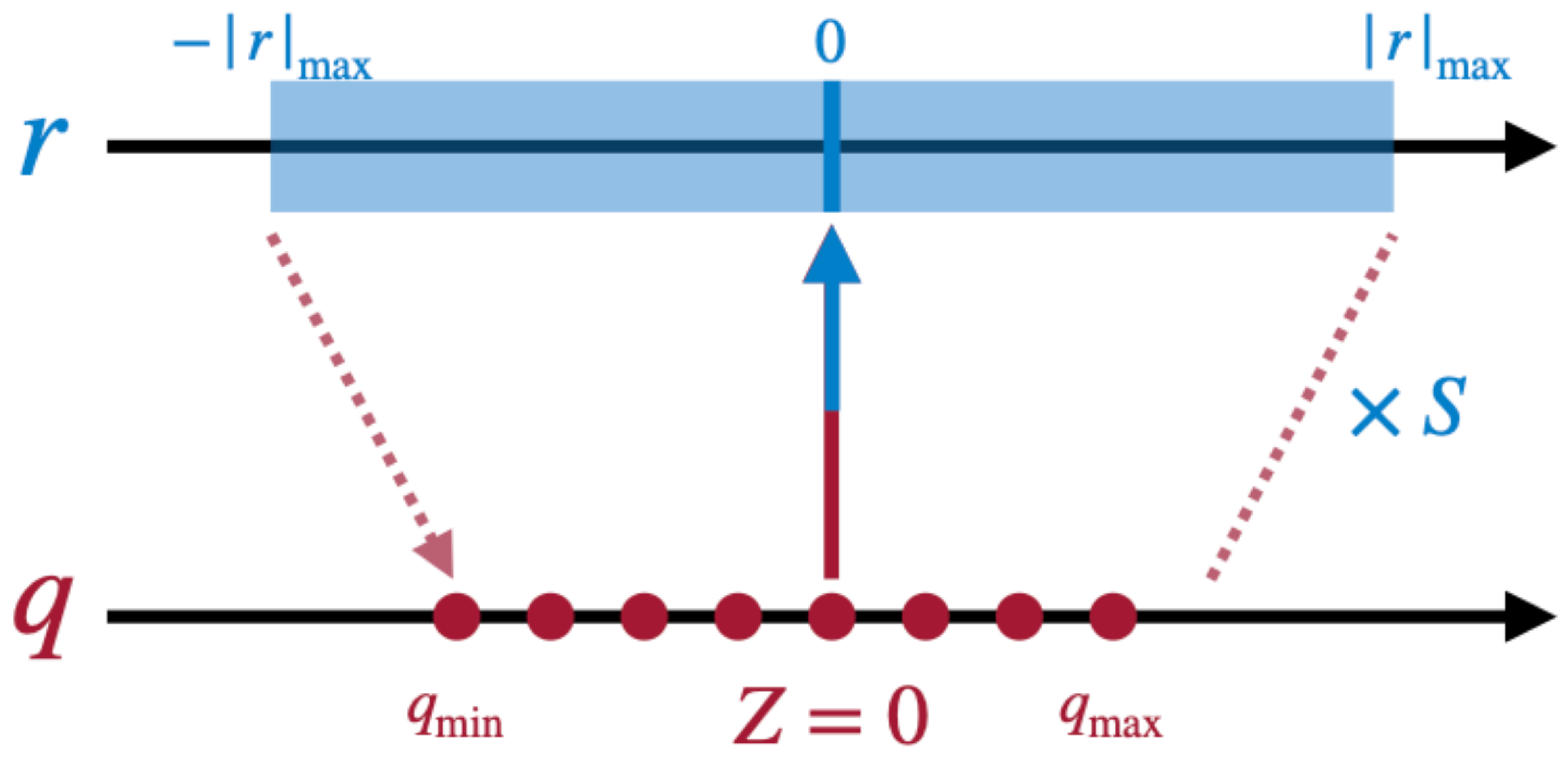
Multiplications done in low-bit integer
Accumulations done in 32-bit integer

Symmetric Q. Uses the range $[-|r|_{\max}, +|r|_{\max}]$ instead of $[r_{\min}, r_{\max}]$



Note. For activations, using ReLU means only nonnegatives will be there, i.e., wasted range.

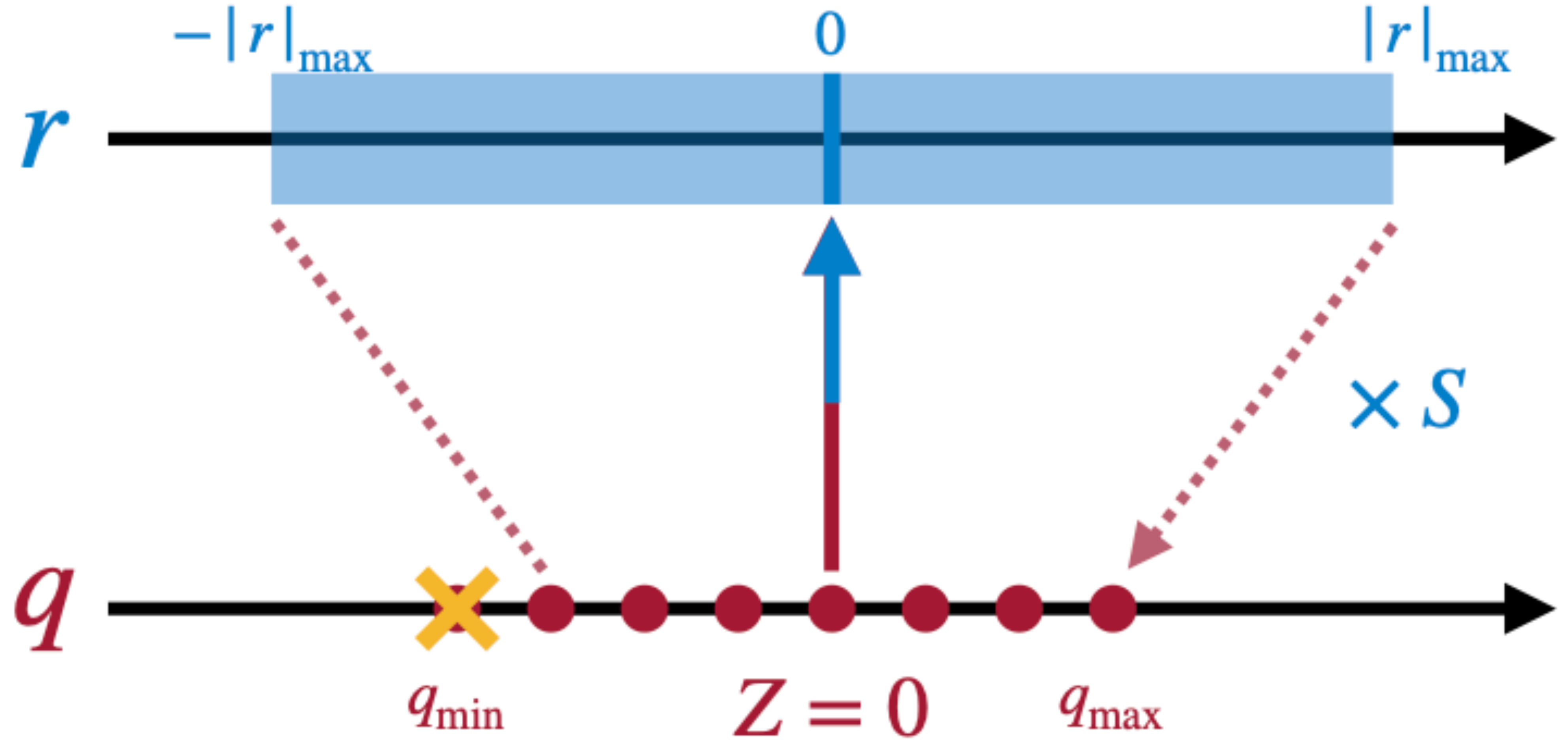
Full-Range. Truncate the max



$$S = \frac{|r|_{\max}}{2^{N-1}}$$

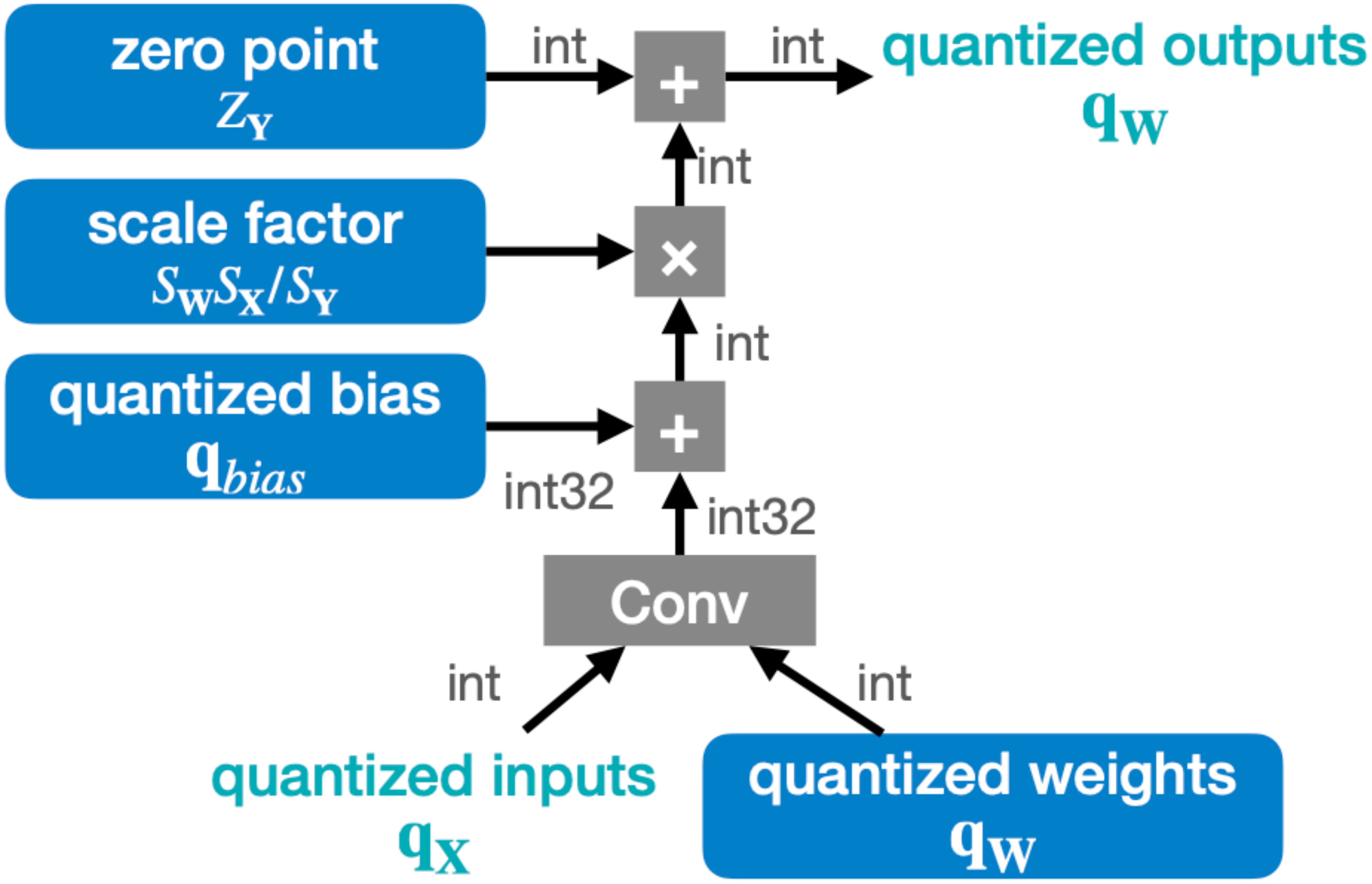
(ONNX, PyTorch, ...)

Restricted-Range. Drop a symbol



$$S = \frac{|r|_{\max}}{2^{N-1} - 1}$$

(TensorFlow, NVIDIA TensorRT, ...)


zero point
$Z_{\mathbf{Y}}$
int
+
int
quantized outputs
$\mathbf{q_W}$
int
scale factor
$S_{\mathbf{W}}S_{\mathbf{X}}/S_{\mathbf{Y}}$
×
int
quantized bias
$\mathbf{q}_{bias}$
+
int32
int32
Conv
int
int
quantized inputs
$\mathbf{q_X}$
quantized weights
$\mathbf{q_W}$