Quantization – 1

EECE695D: Efficient ML Systems

Spring 2025

Agenda

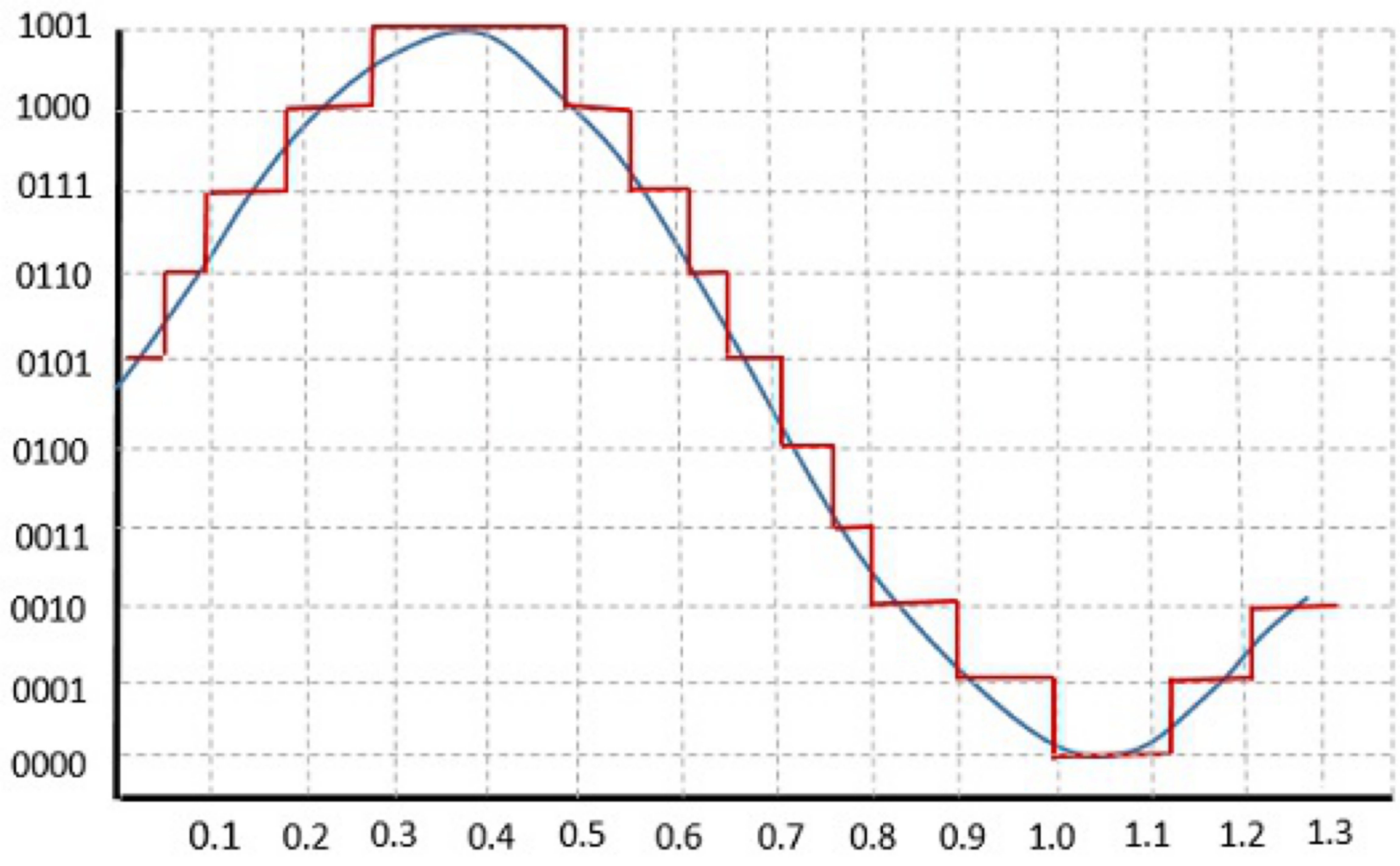
- **Question.** How do we reduce the computational cost of matmuls?
  - W2. Sparsity
    - i.e., reducing the number of nonzero elements
  - W3. Quantization
    - i.e., reducing the precision of weights

- Note. Many graphics from Song Han's lecture notes

Basic idea

Quantization

- Approximating some $X \in \mathscr{X}$ by an element of small, discrete subset $\mathscr{Y} \subseteq \mathscr{X}$
  - $\mathscr{X}$ may be either discrete (e.g., FP32) or continuous (e.g., $\mathbb{R}$)
  - Example. Approximating a float by an integer (e.g., $3.141592 \Rightarrow 3$)



Weight Quantization

- We quantize the weights of a matrix, so that

$$\begin{bmatrix} 2.43 & 1.72 \\ 9.72 & -3.28 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 2 \\ 10 & -3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$$

  - **Memory.** Less bits to store and transfer
  - **Computation**. Less operations to be done
    (as we'll see, it depends on how we quantize)

Activation Quantization

- Plus, we will often do activation quantization (i.e., x)

$$\begin{bmatrix} 2.43 & 1.72 \\ 9.72 & -3.28 \end{bmatrix} \begin{bmatrix} -1.12 & 2.21 \\ 5.27 & 2.09 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 2 & 2 \\ 10 & -3 \end{bmatrix} \begin{bmatrix} -1 & 2 \\ 5 & 2 \end{bmatrix}$$

- Example. If weights and input are integers:
    - Outputs are integers
    - After ReLU, will remain as an integer

$\Rightarrow$ All ops are integers!

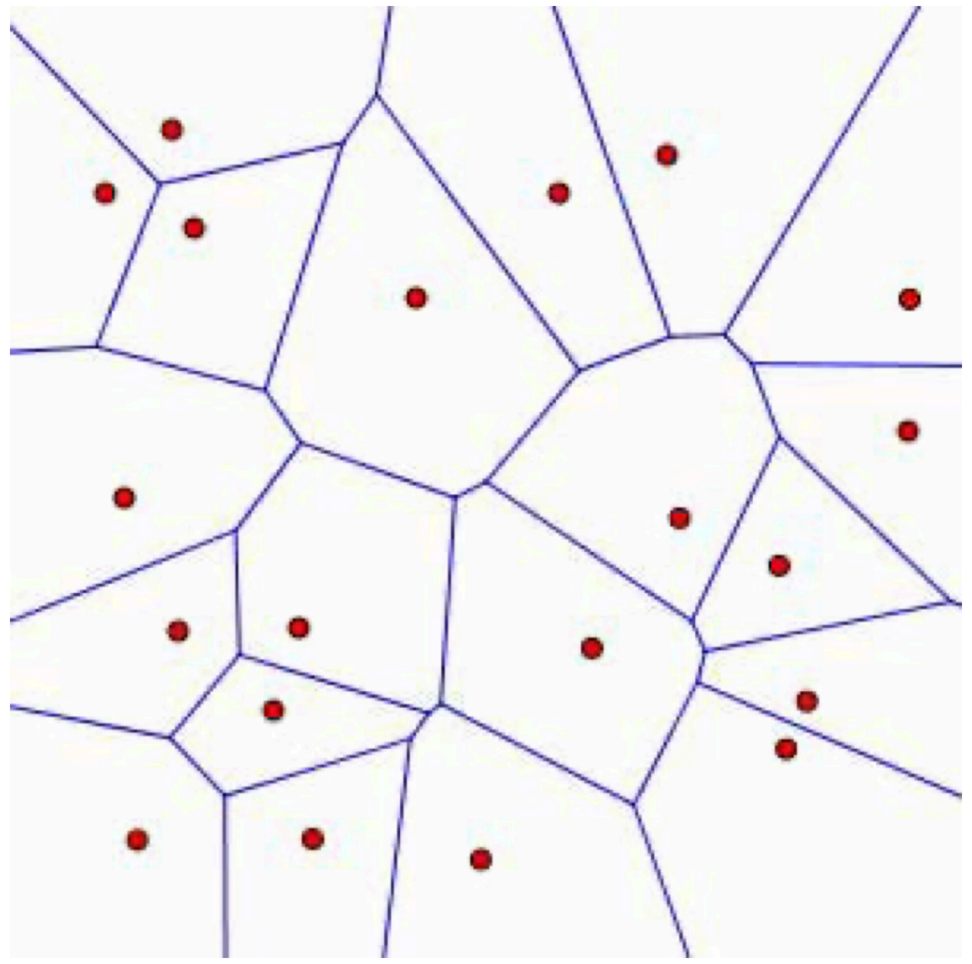
Problem formulation

Problem formulation

- Minimize the training loss of a model with quantized parameters

$$\text{minimize}_{\mathbf{w},q(\cdot)} \quad \hat{L}(q(\mathbf{w}))$$

- Here, $q(\,\cdot\,)$ is a quantization function
    - Applied entrywise (scalar quant.) or blockwise (vector quant.)
        - We assume scalar quantization
    - Different $q(\,\cdot\,)$ is used for each tensor



Problem formulation

- The (scalar) quantization function $q(\,\cdot\,)$ consists of two functions:

$$q = g \circ f$$

  - **Encoder** $f : \mathbb{R} \to \{1,\ldots,K\}$ generates codes from inputs
    - Partitions the space into K groups
  - **Decoder** $g : \{1,\ldots,K\} \to \mathbb{R}$ approximates inputs from codes
    - Decides an output for each partition

(using $\log_2 K$ bits per entry)

Algorithm

Algorithms

$$\text{minimize}_{\mathbf{w},q(\cdot)} \quad \hat{L}(q(\mathbf{w}))$$

- Typically solved by:
  - Train $\mathbf{w}$ (full-precision)
  - Optimize $q(\cdot)$
  - Further tune $\mathbf{w}$ (low-precision)
- Another option: Do quantized training from scratch (later)

Optimizing $q(\cdot)$

- Difficult to optimize using $\hat{L}(\cdot)$
- **Popular.** Relax it to a weight approximation:

$$\text{minimize}_{\mathbf{w}, q(\cdot)} \quad \text{dist}(q(\mathbf{w}), \mathbf{w})$$

  - Here, $\text{dist}(\cdot, \cdot)$ is some distance measure (e.g., ℓ^2 distance)
  - This is equivalent to:

$$\min_{C=\{c_1,\ldots,c_k\}} \min_{\tilde{w}_1,\ldots,\tilde{w}_d \in C} \text{dist}(\tilde{\mathbf{w}}, \mathbf{w})$$

    - C is the "codebook"

Key issue

$$\min_{C=\{c_1,\ldots,c_k\}} \min_{\tilde{w}_1,\ldots,\tilde{w}_d \in C} \text{dist}(\tilde{\mathbf{w}}, \mathbf{w})$$

- A key issue here is to choose the search space of C wisely.
  - **Storage-oriented.** No constraint
    - e.g., K-means quantization
  - **Computation-oriented.** Use HW-friendly data types (e.g., INT)
    - e.g., linear quantization
      - we'll review data types very soon

Another issue

$$\min_{C=\{c_1,\ldots,c_k\}} \min_{\tilde{w}_1,\ldots,\tilde{w}_d \in C} \mathrm{dist}(\tilde{\mathbf{w}}, \mathbf{w})$$

- Of course, this relaxation is not as good as directly minimizing $\hat{L}(\cdot)$
    - Thus we perform further tuning
        - Advanced calibration
        - Quantization-aware training (QAT)
        - (…)

Agenda

- **Today**
  - Recap on data types
  - K-means quantization
- **Next class**
  - Linear quantization
  - Additional tricks

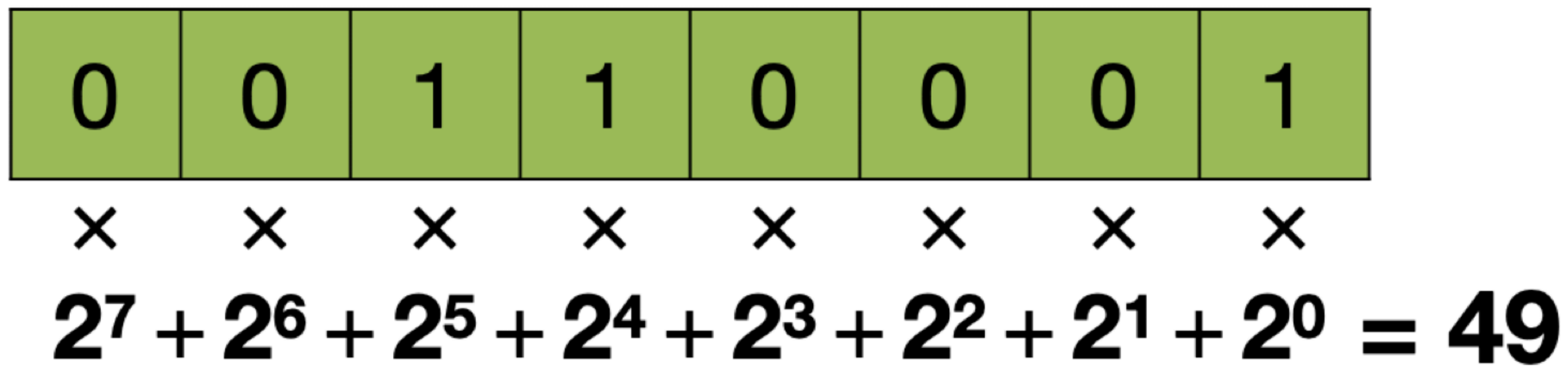
Recap: Data type numerics

Integer (unsigned)

- Given n bits, the value will be computed as

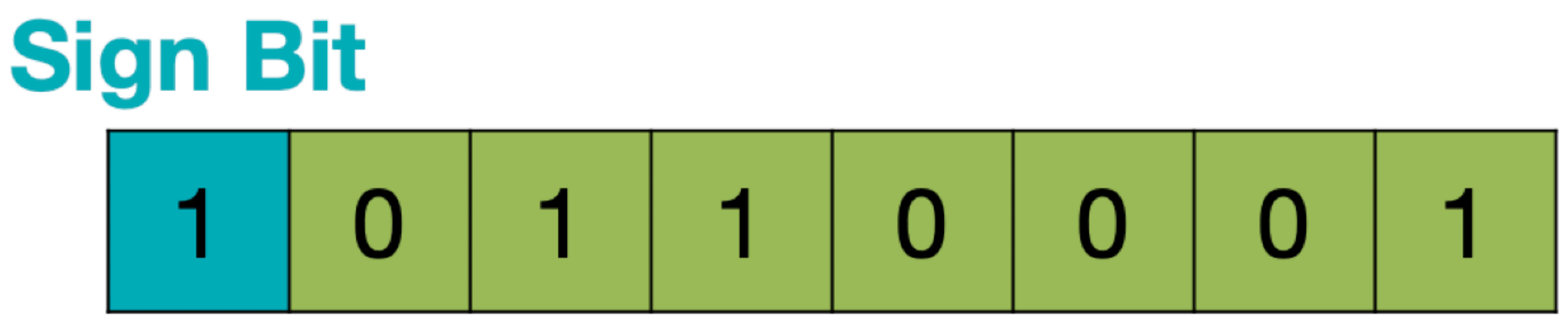
$$\sum_{i=1}^{n} b_i \cdot 2^{n-i}$$

- Covers the range $\{0, \ldots, 2^n - 1\}$



Integer (signed)

- Same as unsigned integer, but uses the **first bit** to represent sign
  - 0: Positive
  - 1: Negative
- Two conventions:
  - Sign-magnitude
  - Two’s complement



INT: Sign-magnitude

- Multiplicative representation of sign
  - First bit represents $\times(-1)$

$$(-1)^{\text{sign}} \quad \times \quad (\text{uint}_{n-1})$$

    - 000...00 denotes zero
    - 100...00 also denotes zero (negative zero; one symbol wasted)
- Covers the range $\{-2^{n-1}-1, \ldots, 2^{n-1}-1\}$

1	0	1	1	0	0	0	1
	×	×	×	×	×	×	×

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

INT: Two's complement

- Additive representation of sign
  - Uses the first bit to represent -2^{n-1}

$$(\text{sign bit}) \cdot (-2^{-n}) \quad + \quad (\text{uint}_{n-1})$$

    - 000...00 denotes 0
    - 100...00 denotes -2^{n-1}
- Covers the range $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$

1	1	0	0	1	1	1	1
×	×	×	×	×	×	×	×

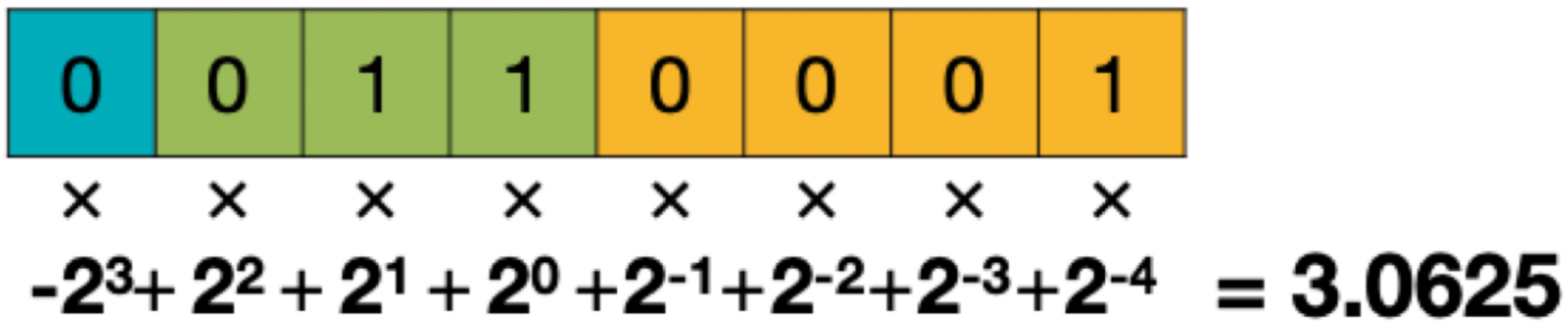
$$-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -49$$

Fixed-point numbers

- Shifts INT by a fixed decimal point

$$(-1)^{\text{sign}} \quad \times \quad (\text{uint}_{n-1}) \quad \times \quad 2^{-\text{d}}$$

- Used for low-cost microprocessors
  - For early uses in DL, see Vanhouke'11, Hwang&Sung'14



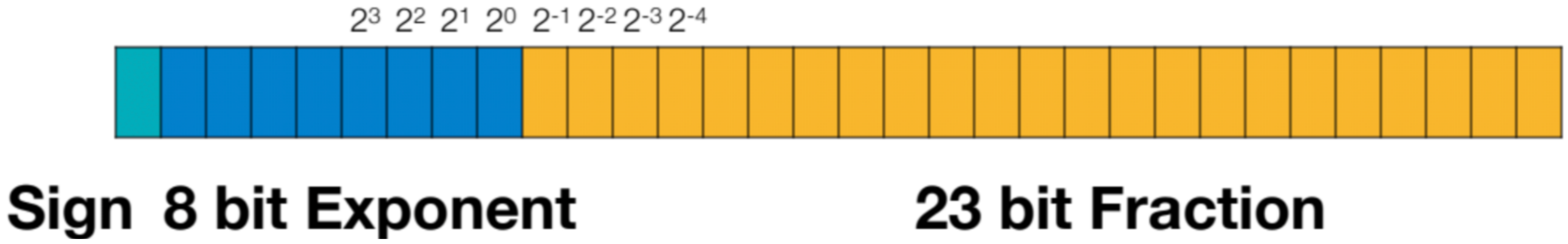
Floating-point numbers

- IEEE 754 standard
- Decimal point is flexibly represented with exponent bits

$$(-1)^{\text{sign}} \quad \times \quad (1 + \text{Fraction}) \quad \times \quad 2^{(\text{Exponent})-127}$$

  - There exists an exponent bias of -127

- **Question.** How do we represent zero?



Floating-point numbers

- Answer. We allocate special symbols to represent end cases
  - If exponent bits are 00...0, we apply a special rule

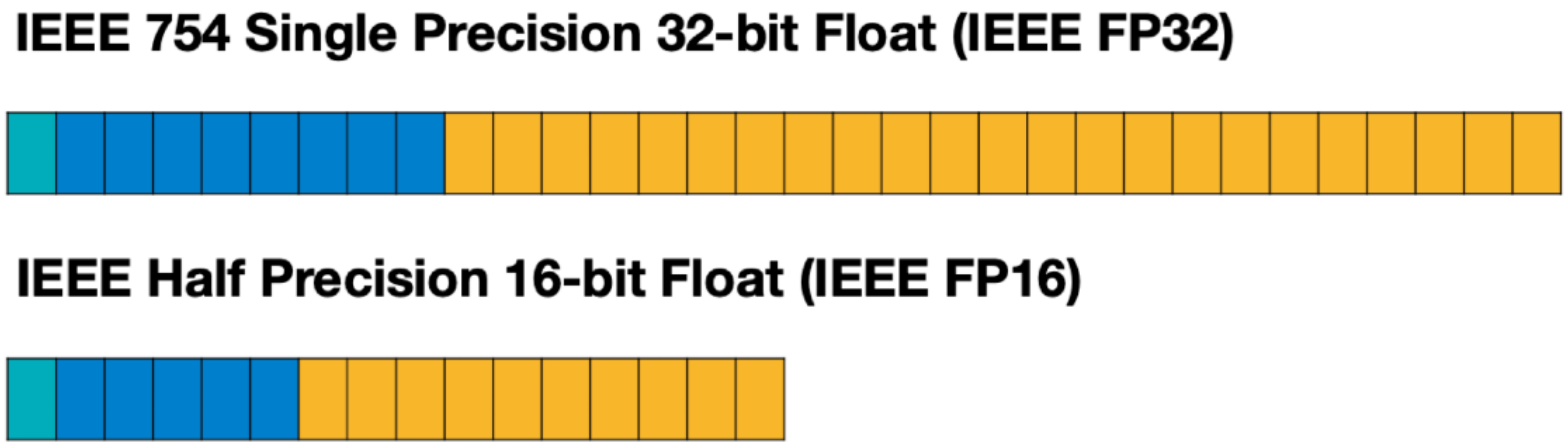
$$(-1)^{\text{sign}} \quad \times \quad (\text{Fraction}) \quad \times \quad 2^{1-127}$$

  - By letting fraction bits be 00...0, we get zero
  - If exponent bias are 11...1, we apply the rules:
    - if fraction bits are 00...0, denotes ∞
    - else, denotes NaN

(wasted bits!)

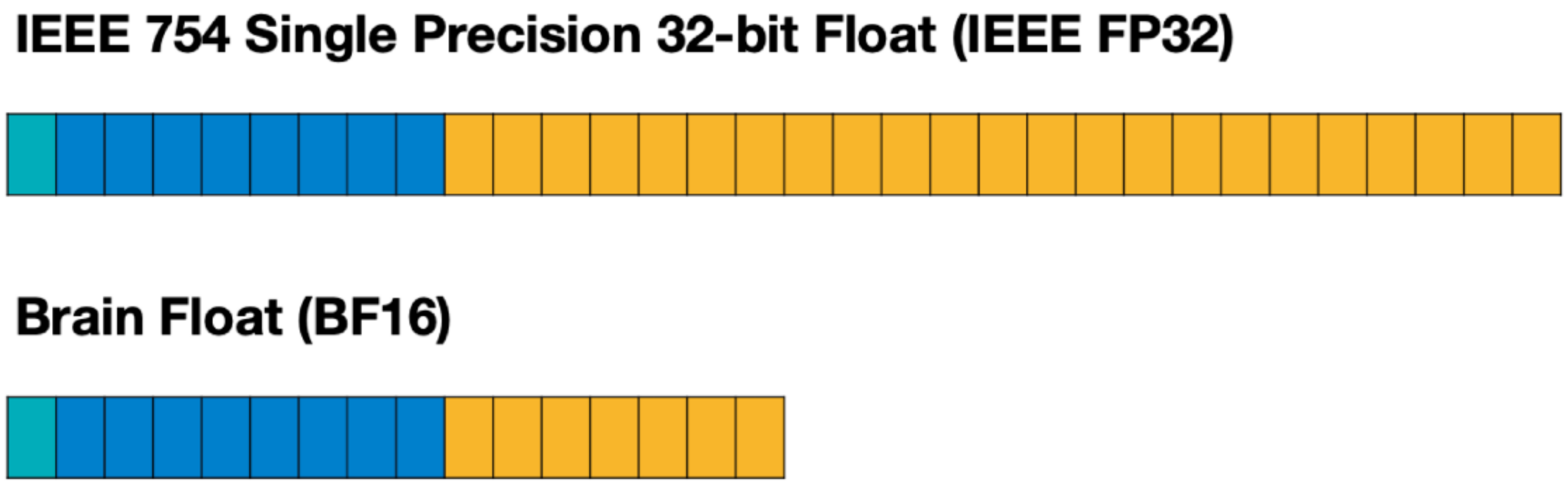
FP16

- FP16 uses less dynamic range and less precision than FP32
  - Exponent: 8 —> 5
  - Fraction: 23 —> 10



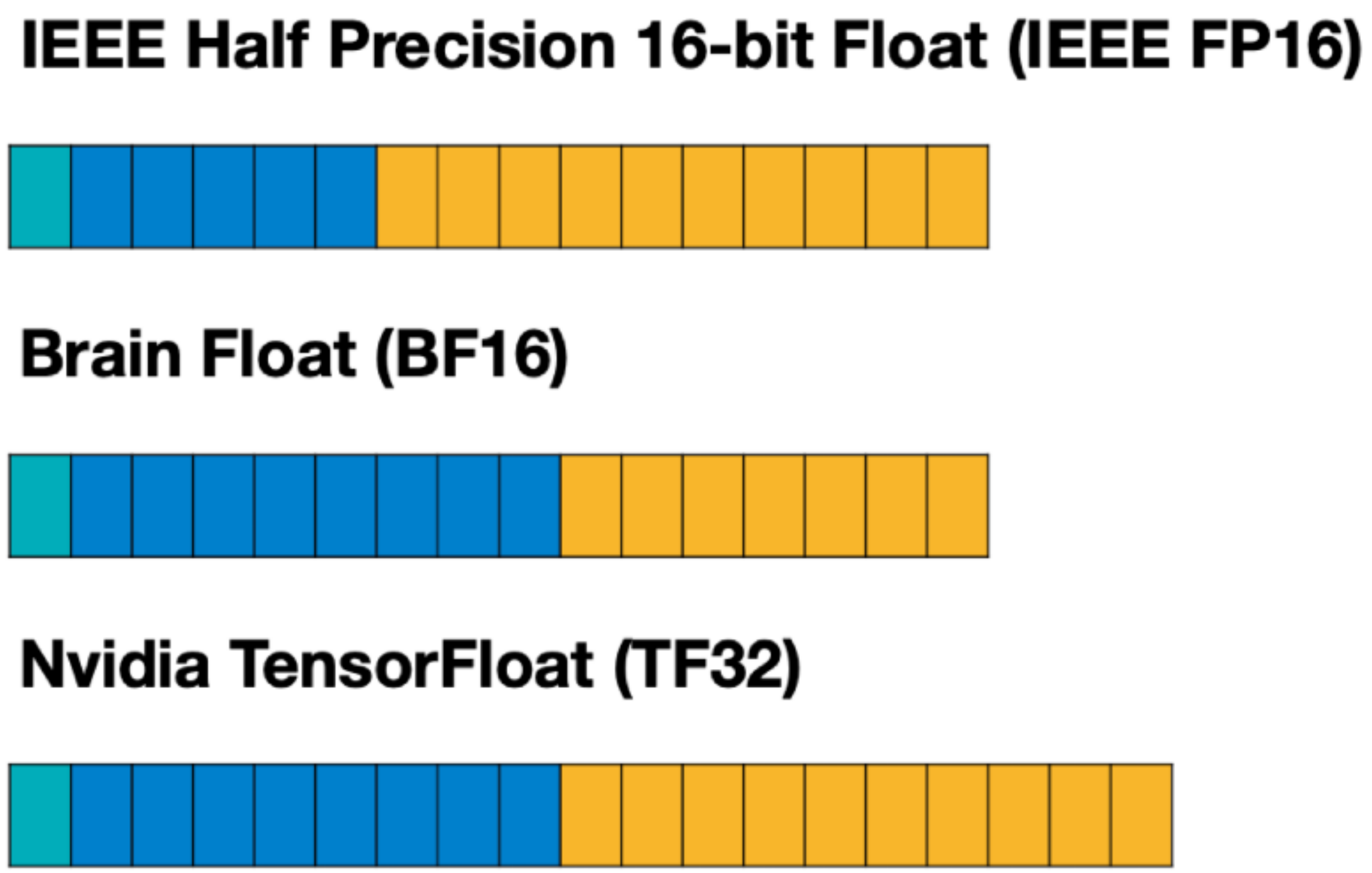
BF16

- Introduced by Google Brain (thus called brain float)
- BF16 uses the same dynamic range and less precision than FP32
    - Exponent: 8 —> 8
    - Fraction: 23 —> 7



TF32

- Introduced in NVIDIA Ampere architectures; stands for“tensor float”
- Uses 19 bits
    - Exponent: 8 (same as BF16)
    - Fraction: 23 —> 10 (same as FP16)



FP8

- Multiple standards
  - Different companies
  - Inference (precision) / Backward (range)
- **Example.** NVIDIA FP8 (in H100)

Nvidia FP8 (E4M3)

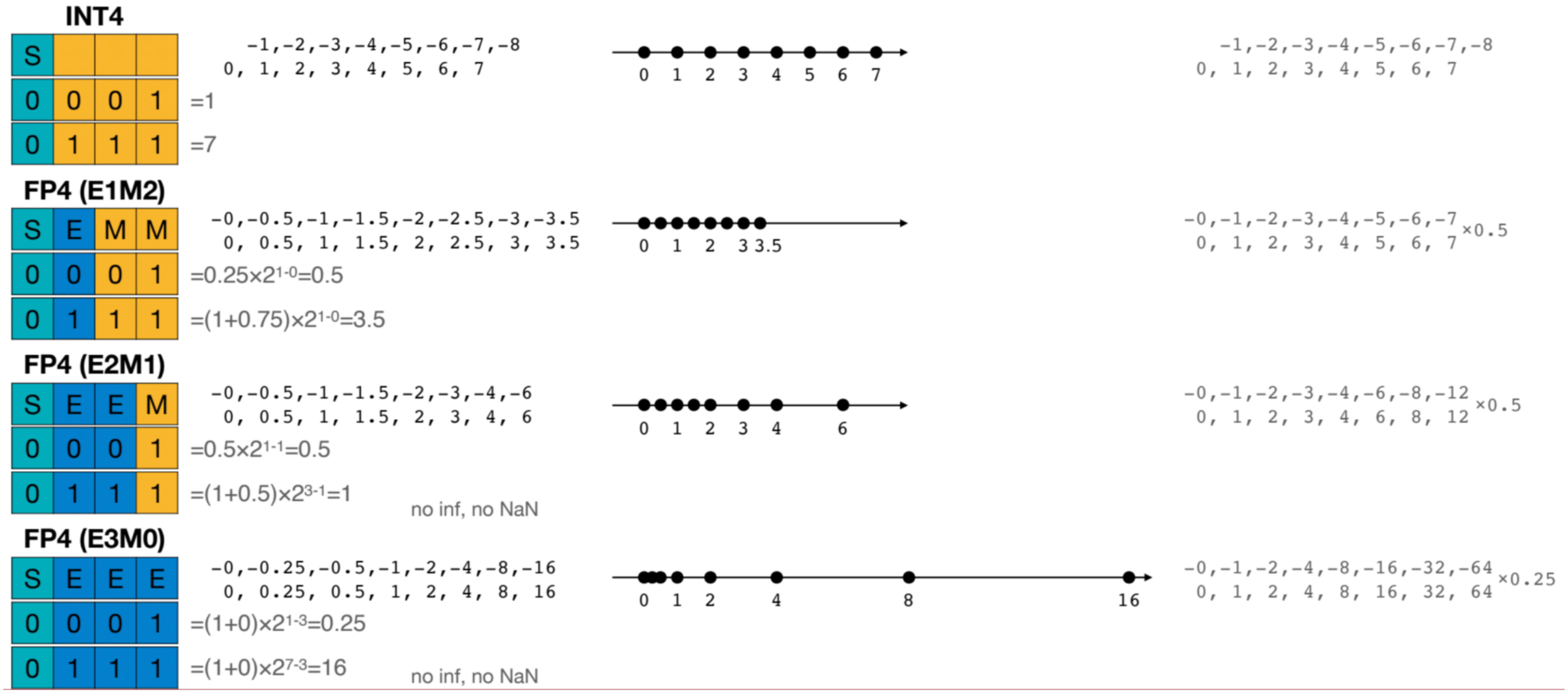
* FP8 E4M3 does not have INF, and $S.1111.111_2$ is used for NaN.
* Largest FP8 E4M3 normal value is $S.1111.110_2 = 448$.

Nvidia FP8 (E5M2) for gradient in the backward

* FP8 E5M2 have INF ($S.11111.00_2$) and NaN ($S.11111.XX_2$).
* Largest FP8 E5M2 normal value is $S.11110.11_2 = 57344$.

FP4

- Very limited dynamic range



FP4

- Yet, the possibility is open
  - Blackwell has added support for FP6, FP4

	Blackwell	Hopper
Supported Tensor Core precisions	FP64, TF32, BF16, FP16, FP8, INT8, FP6, FP4	FP64, TF32, BF16, FP16, FP8, INT8
Supported CUDA® Core precisions	FP64, FP32, FP16, BF16	FP64, FP32, FP16, BF16, INT8

Ampere	Turing	Volta
FP64, TF32, bfloat16, FP16, INT8, INT4, INT1	FP16, INT8, INT4, INT1	FP16
FP64, FP32, FP16, bfloat16, INT8	FP64, FP32, FP16, INT8	FP64, FP32, FP16, INT8

Numerics vs. Throughput

- On A100 GPU math, the relative throughput are:

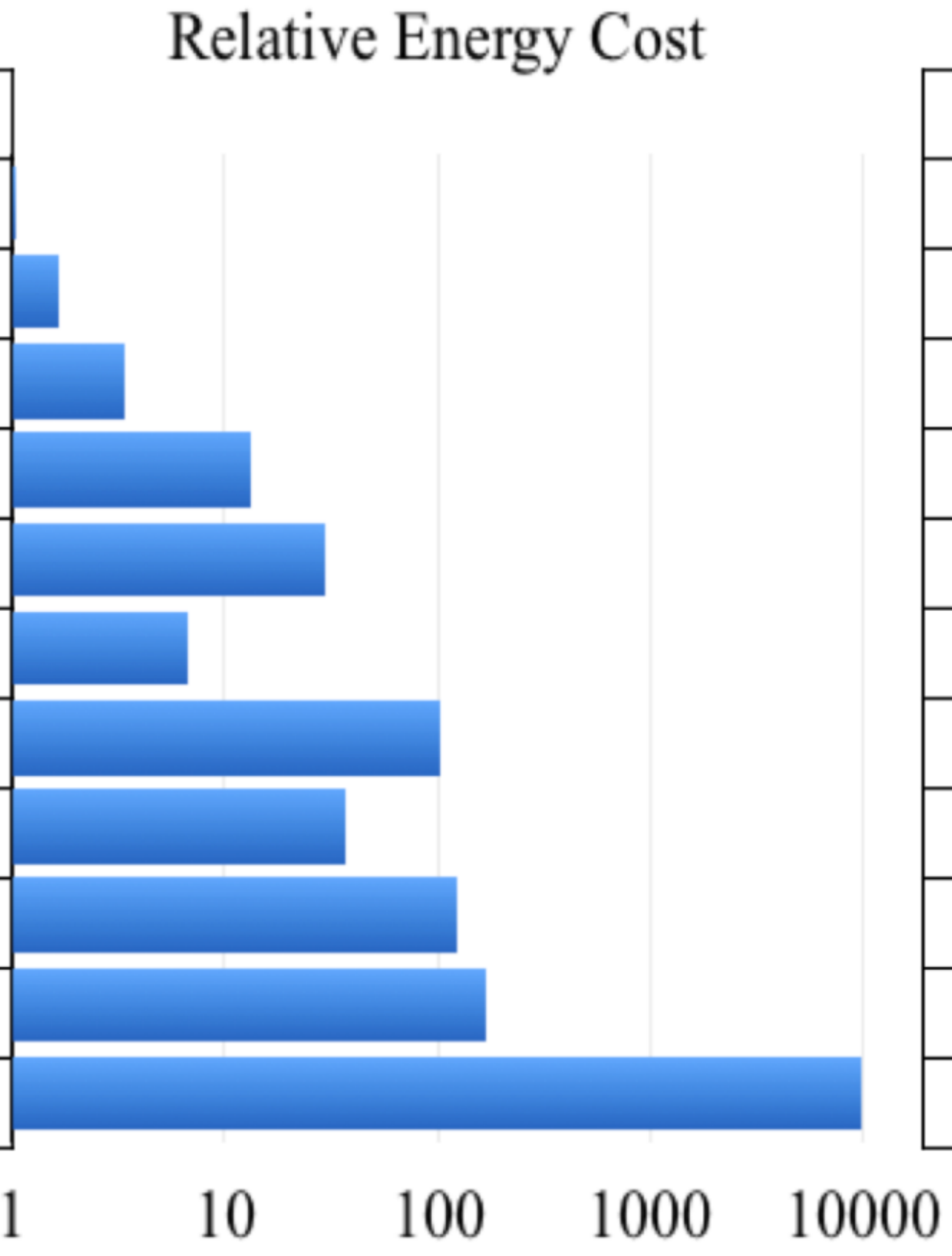
FP32	TF32	FP16 / BF16
1x	8x	16x

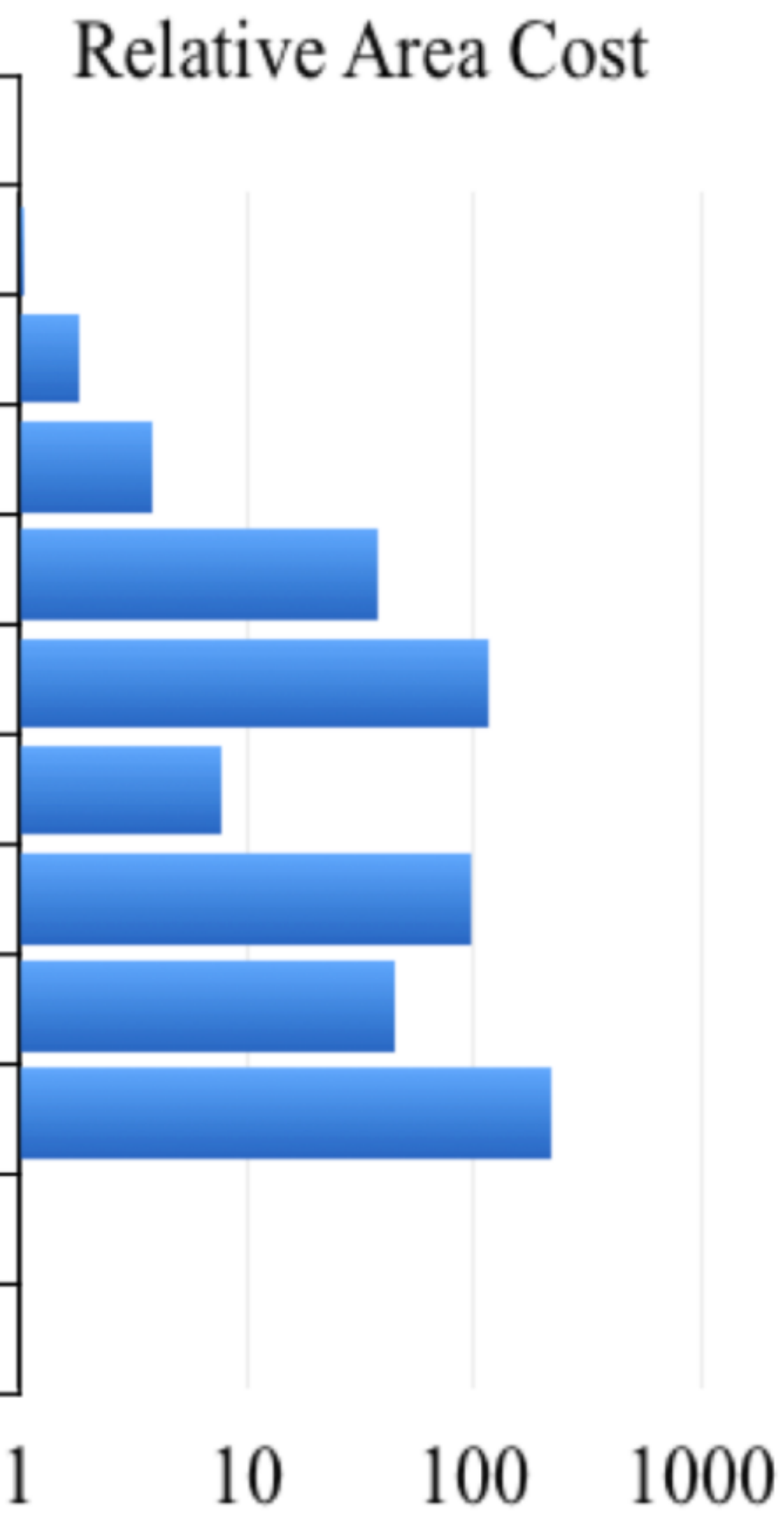
Table 1. Relative throughput of A100 GPU math.

Numerics vs. Energy & Chip area

- On TSMC 45nm 0.9V, different data types and bitwidths translate into:

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A





Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Numerics vs. Training cost

- For certain cases, low-precision training is as good as high-precision
  - Not always true, sadly

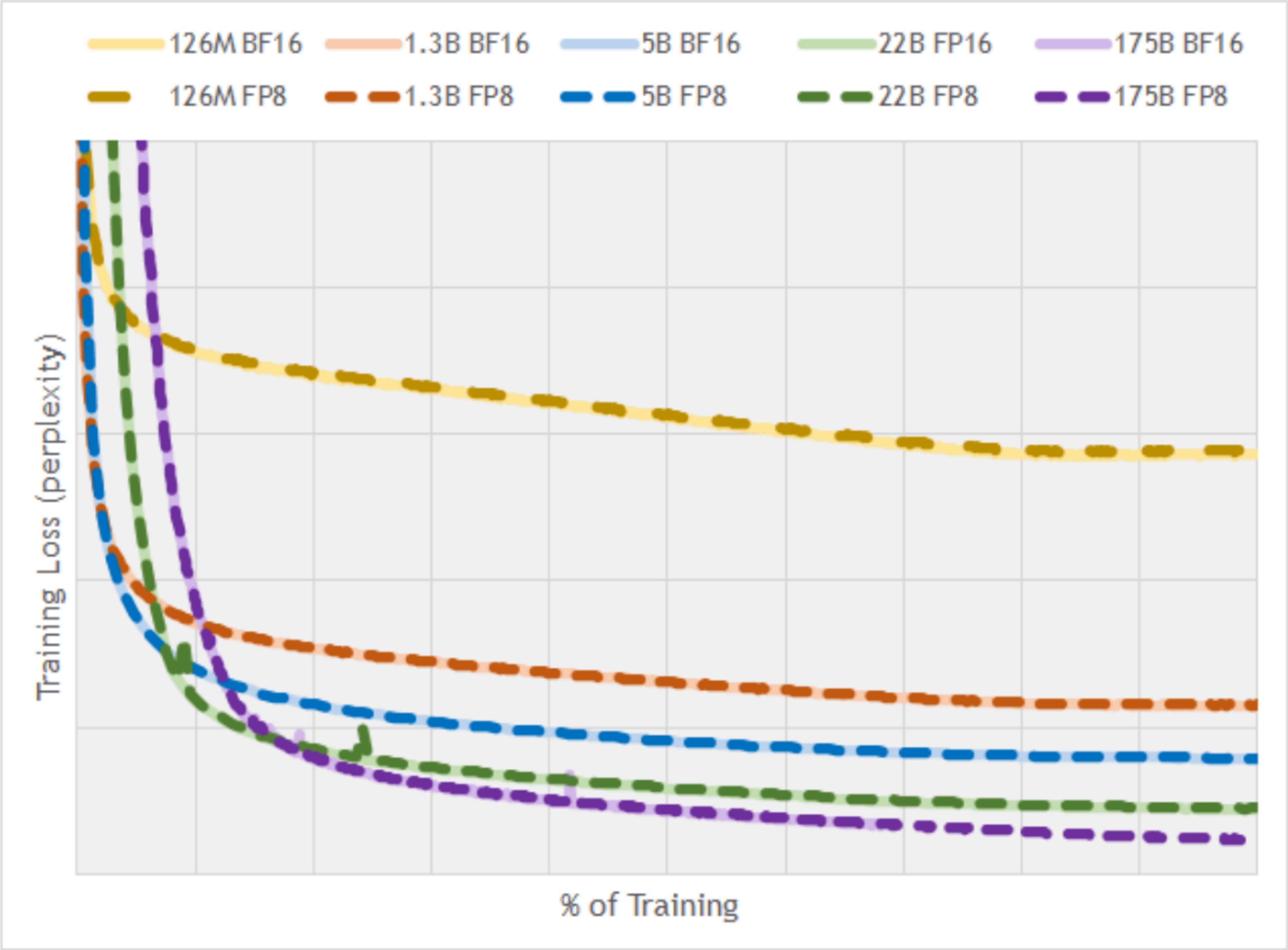


Figure 1: Training loss (perplexity) curves for various GPT-3 models. x-axis is normalized number of iterations.

Micikevicius et al., "FP8 formats for deep learning," arXiv 2022

K-Means Quantization

K-Means Quantization

- Recall that we were solving

$$\min_{C=\{c_1,\ldots,c_k\}} \min_{\tilde{w}_1,\ldots,\tilde{w}_d \in C} \mathrm{dist}(\tilde{\mathbf{w}}, \mathbf{w})$$

- K-means quantization puts no constraint on C:
  - **Storage.** Well optimized
  - **Computation.** Cannot use low-bit matmuls
    - Plus, requires weights to be decoded to full-precision before use

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

Algorithm

- K-means quantization simply use ℓ^2 distance:

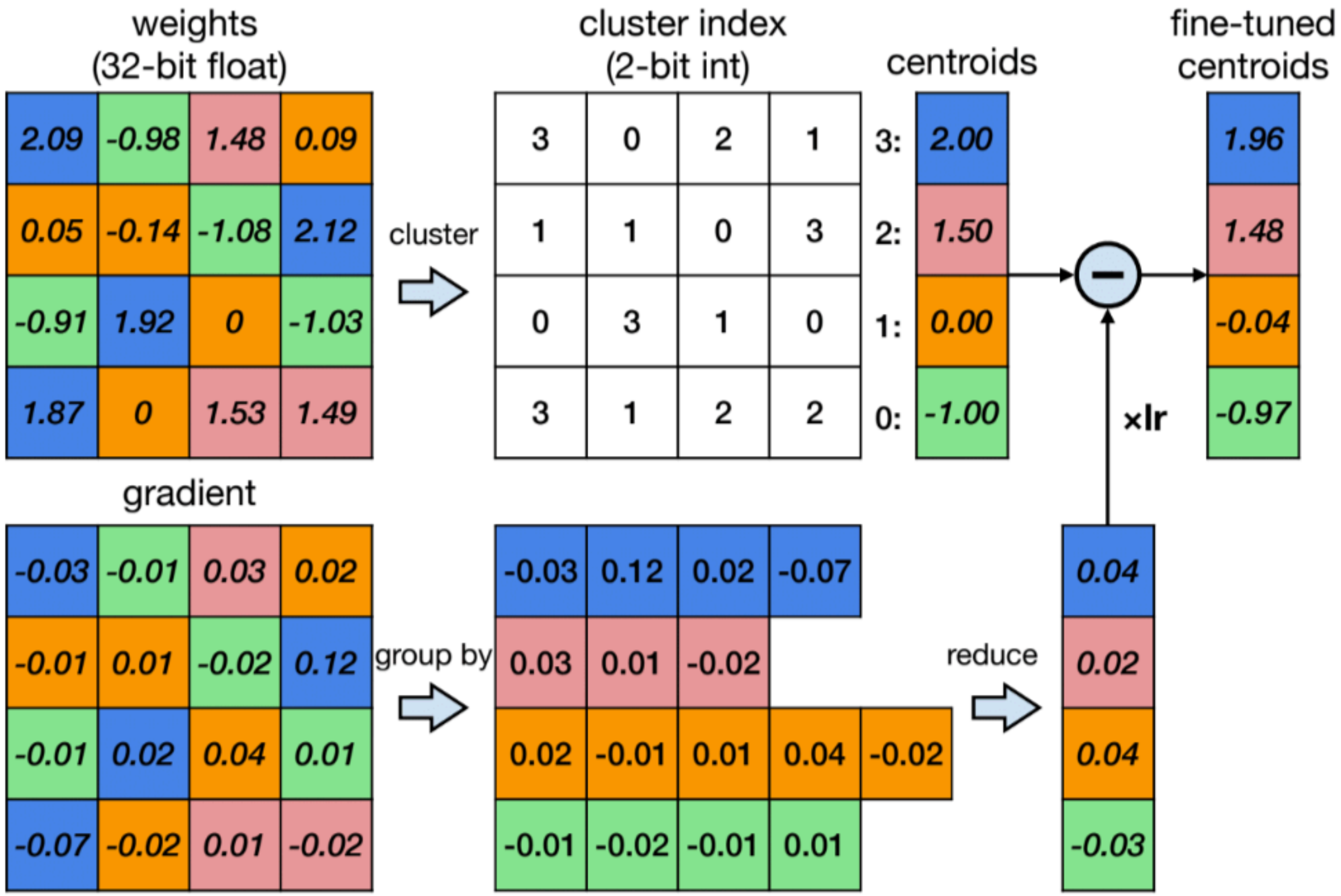
$$\min_{C=\{c_1,\ldots,c_k\}} \min_{\tilde{w}_1,\ldots,\tilde{w}_d \in C} \sum_{i=1}^{d} (\tilde{\mathbf{w}}_i - \mathbf{w}_i)^2$$

- This is exactly 1D K-means, with neural network weights as the data.
    - Solved via Lloyd's algorithm
        - Assign weights to clusters by nearest neighbor matching
        - Compute centroids via averaging
        - Repeat until convergence

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

Algorithm

- As ℓ^2 loss is imperfect, we fine-tune the centroids using the average gradients of the weights assigned to each cluster

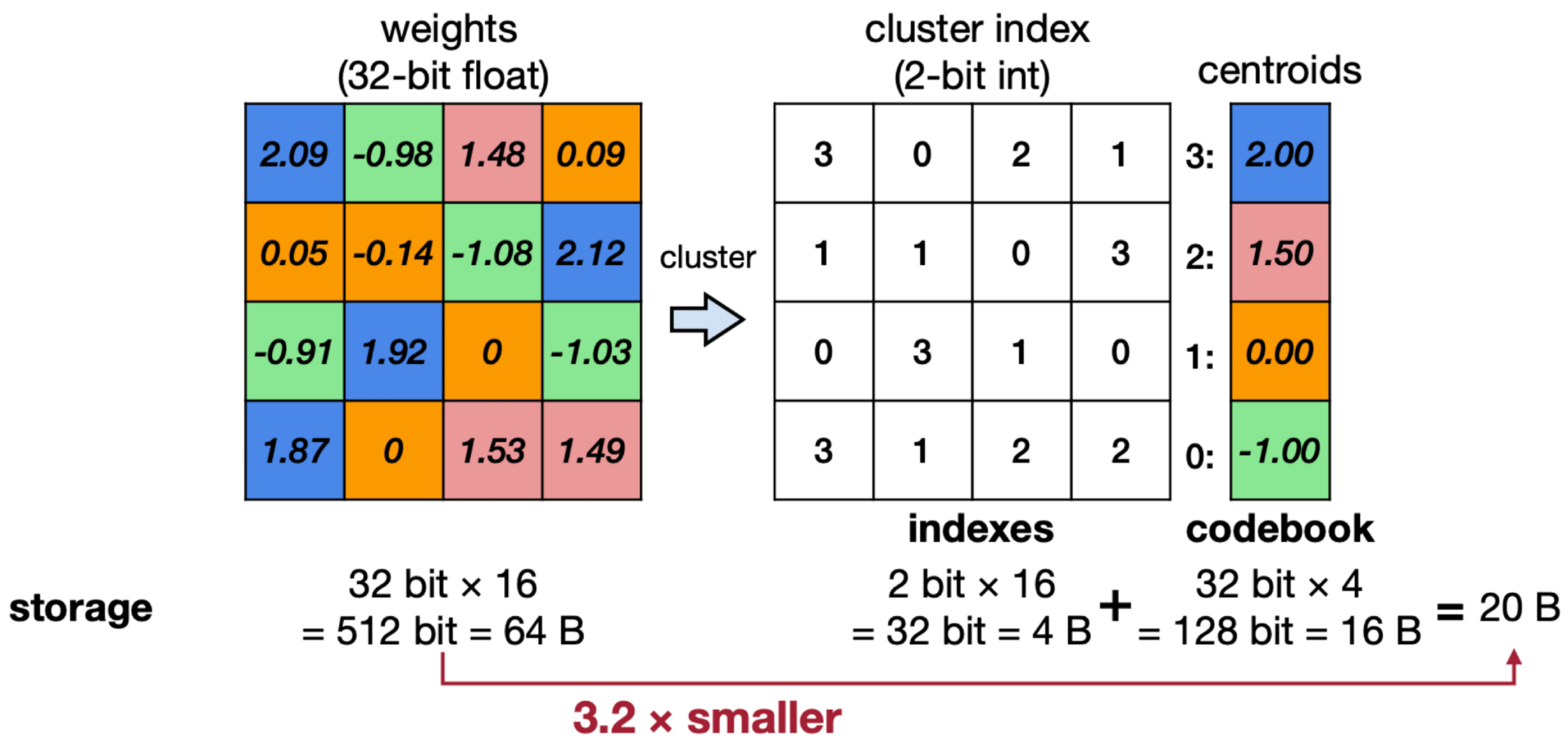


Han et al., "Deep Compression: Compressing DNNs with pruning, trained quantization, and Huffman coding," ICLR 2016

Storage

- **Note.** We need to store the codebook as well!
  - If we quantize $N \times N$ matrix with codebook size K, the compression rate is

$$\frac{(\log_2 K)N^2/8 + 4K}{4N^2}$$



Next Class

- Linear quantization
- Various issues
    - Granularity
    - Rescaling
    - Clipping
    - Rounding
    - QAT

That’s it for today 🙌