



Accelerator Architectures for Machine Learning (AAML)

Lecture 7: Low-bit Quantization

Tsung Tai Yeh

Department of Computer Science
National Yang-Ming Chiao Tung University



Acknowledgements and Disclaimer

- Slides was developed in the reference with
Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019 tutorial
Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin Chen,
Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
Yakun Sophia Shao, EE290-2: Hardware for Machine Learning, UC
Berkeley, 2020
CS231n Convolutional Neural Networks for Visual Recognition, Stanford
University, 2020
- 6.5940, TinyML and Efficient Deep Learning Computing, MIT
- NVIDIA, Precision and performance: Floating point and IEEE 754
Compliance for NVIDIA GPUs, TB-06711-001_v8.0, 2017



Outline

- Low-bit Precision Formats
 - NVFP4, MXFP8
- K-Means-based Quantization
- Linear Quantization
- Binary and Ternary Quantization



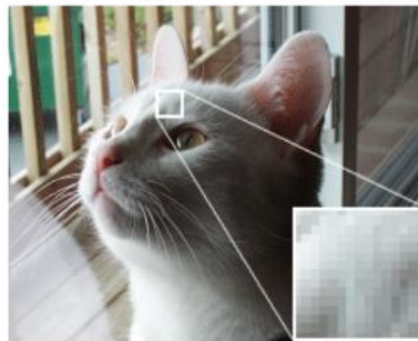
What is Quantization ?

- **Quantization**

- A process that reduces the precision of a digital signal by converting high-precision data into a lower-precision format



Original Image



16-Color Image

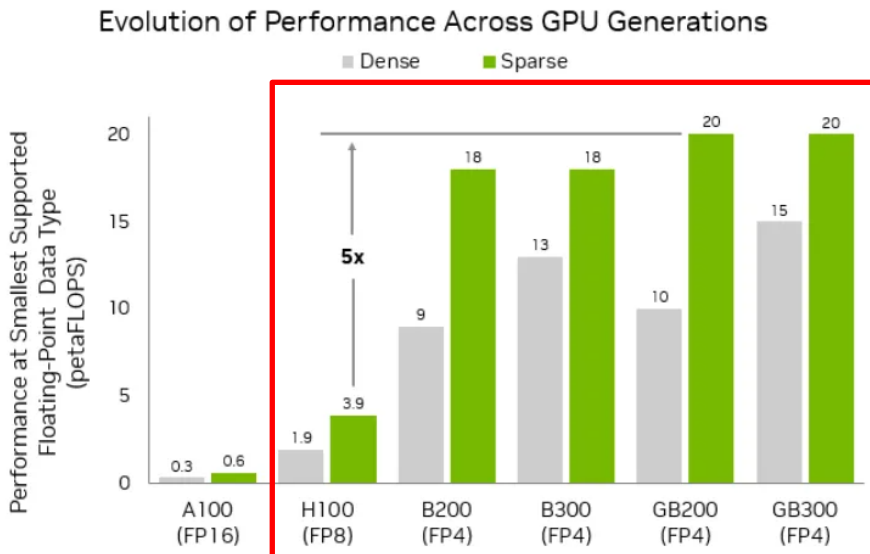


Images are in the public domain.



Benefits of Quantization

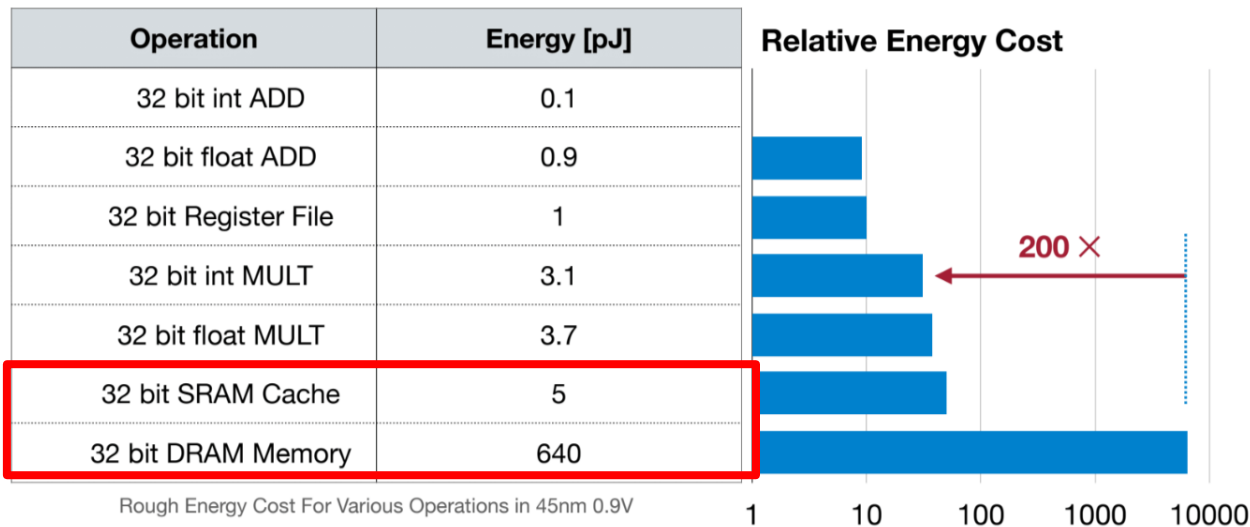
- **Reduced memory burden**
 - Reduce pressure on memory bandwidth which can improve output token throughput
- **Simplified compute operations**
 - Improve overall end-to-end latency performance as a result of simplified attention layer computations





Memory is Expensive !!

- **Data movement -> Move memory reference -> More energy**



1  = 200 X +

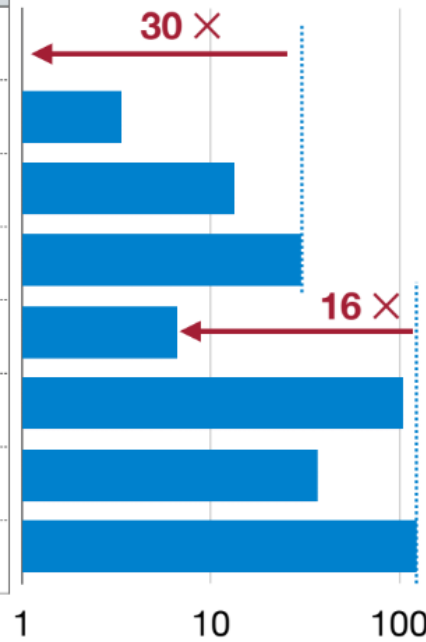
[This image](#) is in the public domain



Low Bit-Width Operations are Cheap

- **Less Bit-Width -> Less energy**

Operation	Energy [pJ]
8 bit int ADD	0.03
32 bit int ADD	0.1
16 bit float ADD	0.4
32 bit float ADD	0.9
8 bit int MULT	0.2
32 bit int MULT	3.1
16 bit float MULT	1.1
32 bit float MULT	3.7



45 nm Process, Horowitz,
ISSCC, 2014



Energy and Area Cost

Could we make the deep learning efficient by lowering the precision of data ?

Operation	Energy (pJ)	Area(um 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
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	
32b DRAM Read	640	

173X

4.7X



Numeric Data Types

- Fixed-point number



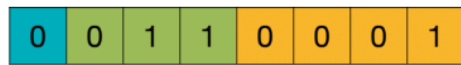
Integer . Fraction

“Decimal” Point



x x x x x x x x

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



x x x x x x x x

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

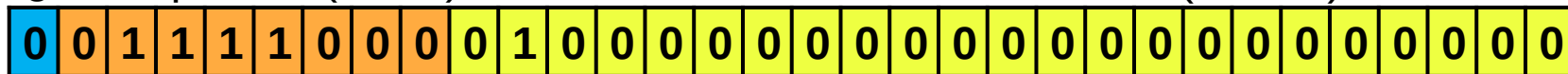


IEEE 765 Single Precision Float Point

- **Sign** determines the sign of the number
- **Exponent** (8 bit) represent -127 (all 0s) and +128 (all 1s)
- **Significand** (23 fraction bits), total precision is 24 bits (23 + 1 implicit leading bit) $\log_{10}(2^{24}) \approx 7.225$ digital bit

Sign Exponent (8 bits)

Mantissa/Fraction (23 bits)



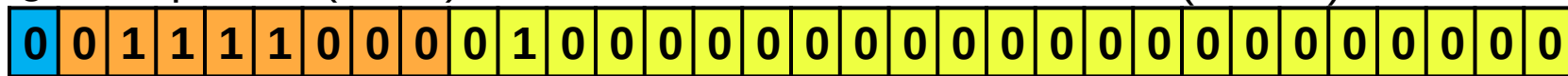
$$value = (-1)^{sign} \times 2^{(e-127)} \times \left(1 + \sum_{i=1}^{23} b_{(23-i)} 2^{-i}\right)$$



IEEE 765 FP32

Sign Exponent (8 bits)

Mantissa/Fraction (23 bits)



$$value = (-1)^{sign} \times 2^{(e-127)} \times (1 + \sum_{i=1}^{23} b_{(23-i)} 2^{-i})$$

$$\text{Sign} = b_{31} = 0 ; (-1)^0 = 1$$

$$e = 120; 2^{(120 - 127)} = 2^{-7}$$

$$1.b_{22}b_{21}...b_0 = (1 + \sum_{i=1}^{23} b_{(23-i)} 2^{-i}) = 1 + 2^{-2} = 1.25$$

$$\text{Value} = 1 \times 2^{-7} \times 1.25 = 0.009765625$$



Numeric Data Type

- **Question:** What is the decimal “**11.375**” in FP32 format ?

11.375

= 11 + 0.375

= $(1011)_2 + (0.011)_2$

= $(1011.011)_2$

= $(1.011011)_2 \times 2^3$

$0.375 \times 2 = 0.750 = 0 + 0.750 \Rightarrow b_{-1} = 0$

$0.750 \times 2 = 1.500 = 1 + 0.500 \Rightarrow b_{-2} = 1$

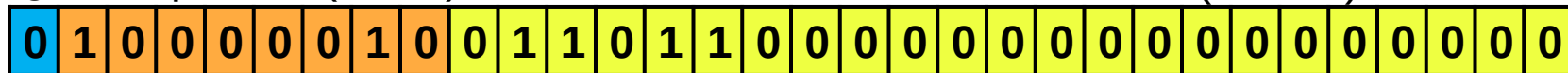
$0.500 \times 2 = 1.000 = 1 + 0.000 \Rightarrow b_{-3} = 1$

- The exponent is 3 and biased form

= $(3 + 127) = 130 = 1000\ 0010$

Sign Exponent (8 bits)

Mantissa/Fraction (23 bits)





Floating-Point Number

- Exponent Width -> Range; Fraction Width-> Precision

IEEE 754 Single Precision 32-bit Float (IEEE FP32)



IEEE Half Precision 16-bit Float (IEEE FP16)



Brain Float (BF16)



Nvidia TensorFloat (TF32)



AMD 24-bit Float (AMD FP24)



Exponent (bits)	Fraction (bits)	Total (bits)
8	23	32
5	10	16
8	7	16
8	10	19
7	16	24

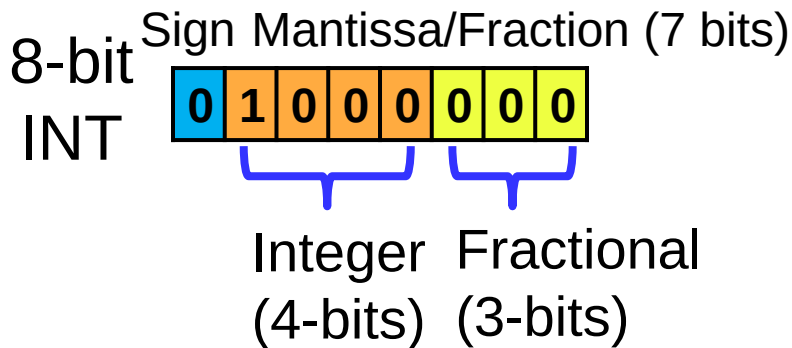
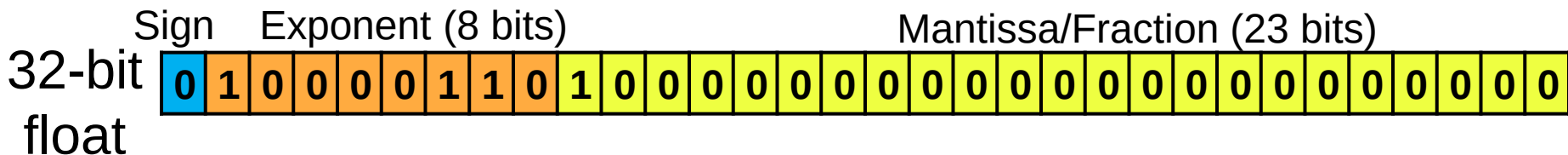


Number Representation

	1	8	23	
FP32	S	E	M	
	1	5	10	
FP16	S	E	M	
	1	31		
INT32	S	M		
	1	15		
INT16	S	M		
	1	8		
INT8	S	M		
				Range
				1.2E-38 to 3.4E+38
				6.1E-5 to 6.6E+4
				2147483648 to 2147483647
				-32,768 to 32,767
				-128 ~ 127



Reduced Bit Width





FP32 vs FP16 vs BF16

- **FP32 – single precision**
 - With 6-9 significant decimal digits precision
- **FP16 – half precision**
 - Uses in some neural network applications
 - With 4 significant decimal digits precision
- **BF16**
 - A truncated FP32
 - Allow for fast conversion to and from an FP32
 - With 3 significant decimal digits

(a) fp32: Single-precision IEEE Floating Point Format

Range: $\sim 1e^{-38}$ to $\sim 3e^{38}$



(b) fp16: Half-precision IEEE Floating Point Format

Range: $\sim 5.96e^{-8}$ to 65504



(c) bfloat16: Brain Floating Point Format

Range: $\sim 1e^{-38}$ to $\sim 3e^{38}$



Format	Bits	Exponent	Fraction
FP32	32	8	23
FP16	16	5	10
BF16	16	8	7



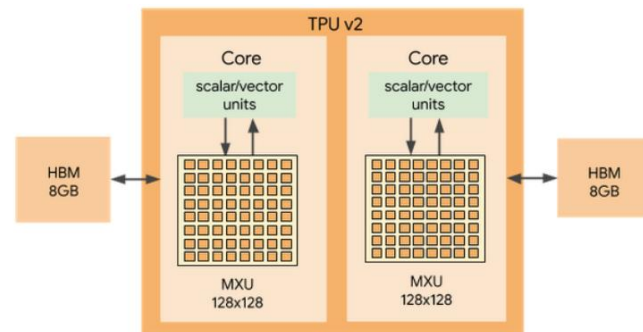
Choosing bFloat16

- **Motivation**

- The physical size of a hardware multiplier scales with the square of the mantissa width
- Mantissa bit length – FP32-> 23 bits, FP16-> 10 bits, BF16:->7 bits

- **BF16**

- 8 X smaller than an FP32 multiplier
- Has the same exponent size as FP32
- No require special handling (loss scaling) in the FP16 conversion
- XLA compiler's automatic format conversion
- In side the MXU, multiplications are performed in BF16 format
- Accumulations are performed in full FP32 precision

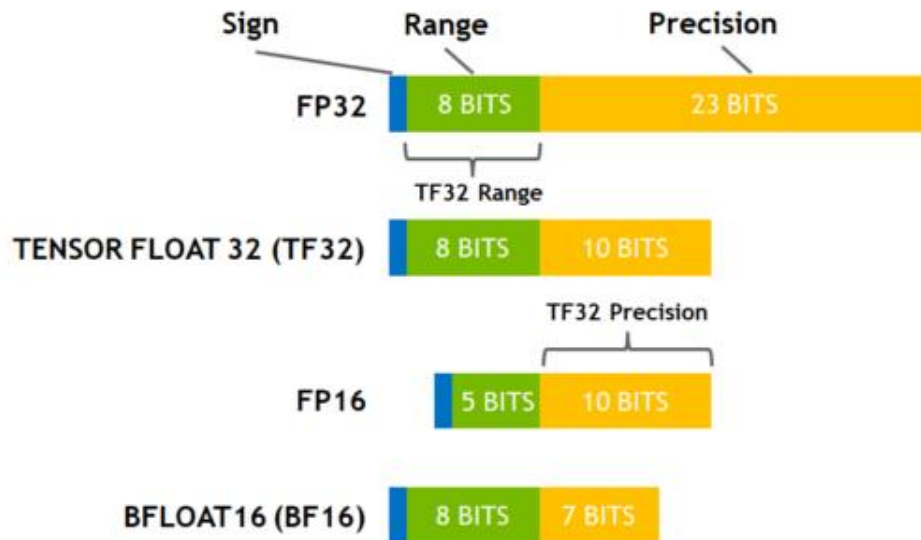




Nvidia's TF32

- **Nvidia's TF32**

- 19-bit (BF19)
- 1-bit sign, 8-bit exponent
10-bit fraction
- Fuse BF16 and FP16
 - BF16: 8-bit exponent +
 - FP16: 10-bit fraction
- Nvidia A100 Tensor Core
 - TF32: 156 TFLOPS
 - FP16/BF16: 312 TFLOPS



<https://reurl.cc/Omo1dv>



FP8 and Tesla CFloat

- **FP8 (1-5-2)**

- Large loss in MobileNet v2
- Hybrid FP8 (HFP8)
 - Use FP(1-4-3) in forward
 - Use FP(1-5-2) in backward

- **Tesla Dojo Cfloat (configurable float)**

- Configurable exponent and mantissa
- Use software to choose appropriate Cfloat format
 - CF16
 - CF8 (1-4-3), CF8 (1-5-2)

c. Trans-precision Inference Accuracy of FP32 models in FP8 1-5-2 precision

FP32 Model	Baseline	FP8 1-5-2
MobileNet_v2 ImageNet	71.81	52.51
ResNet50 ImageNet	76.44	75.31
DensetNet121 ImageNet	74.76	73.64
MaskRCNN COCO [†]	33.58 29.27	32.83 28.65

[†] Box and Mask average precision

<https://proceedings.neurips.cc/paper/2019/file/65fc9fb4897a89789352e211ca2d398f-Paper.pdf>



Nvidia's NVFP4

- **Nvidia's NVFP4**

- 1 sign bit, 2 exponent bits, and 1 mantissa bit (E2M1)
- The value in the format ranges approximately -6 to 6
- The values in the range could include 0.0, 0.5, 1.0, 1.5, 2, 3, 4, 6 (same for the negative range)

Table 1: Numbers represented in FP4-E2M1 with NaN and Inf (IEEE 754 standard) and Numbers represented in FP4-E2M1 without NaN and Inf (Our design).

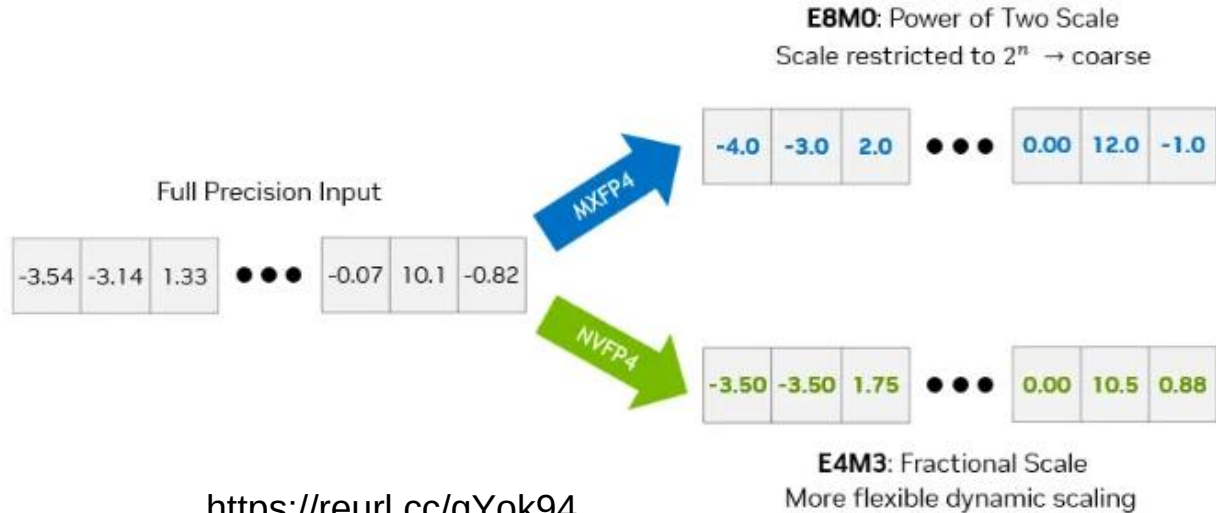
UINT4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FP4 w/ NaN&Inf	0	0.5	1	1.5	2	3	Inf	NaN	-0	-0.5	-1	-1.5	-2	-3	Inf	NaN
FP4 w/o NaN&Inf	0	0.5	1	1.5	2	3	4	6	-0	-0.5	-1	-1.5	-2	-3	-4	-6



Nvidia's NVFP4

- **High-precision scaling**

- NVFP4 encodes blocks using E4M3 FP8 precision
- Enables non-power-of-two scaling factors with fractional precision

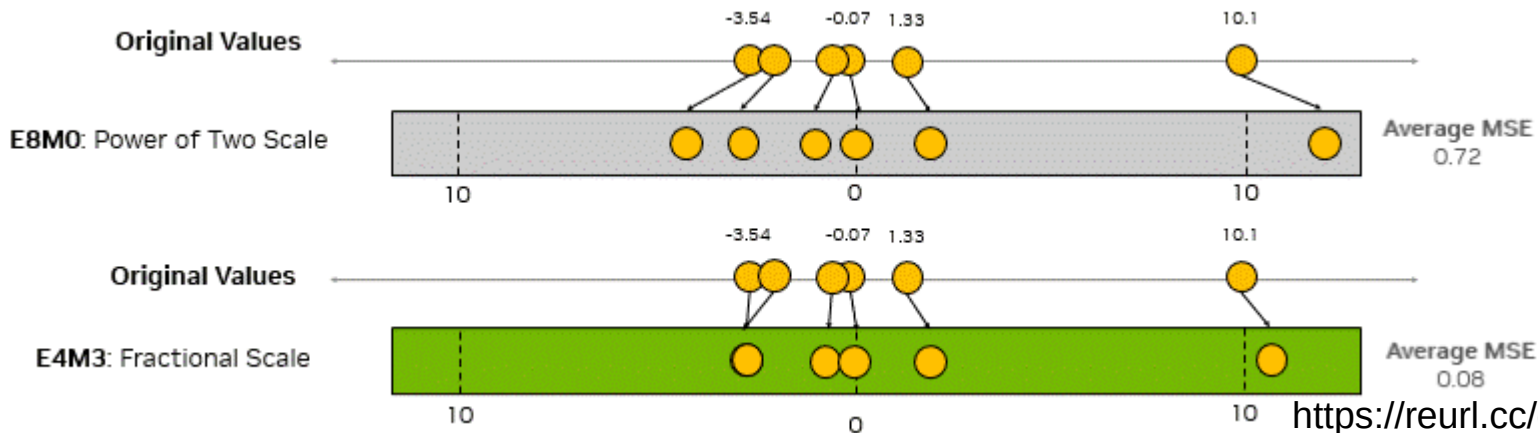




Nvidia's NVFP4

- **High-precision scaling**

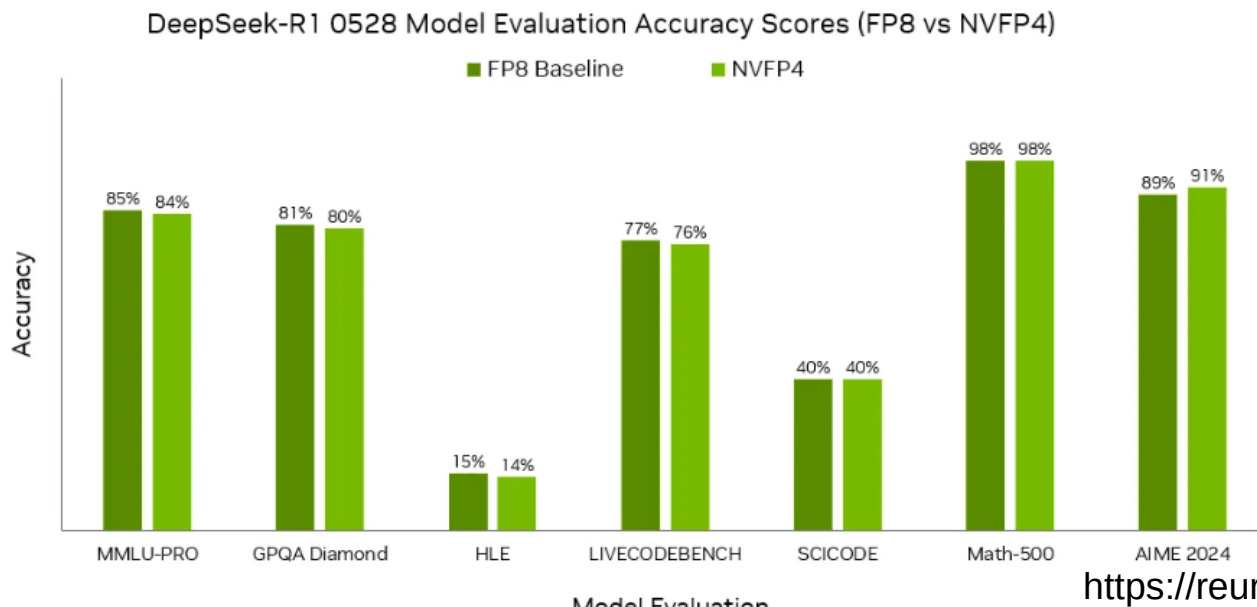
- E8M0 = Snaps the scale factor to nearest 2^n
- E4M3 = Finds one scale factor that makes the block errors collectively as small as possible





Nvidia's NVFP4

- **Quantize model weights to 4-bits**
 - The analysis showcases the 1% or less accuracy degradation





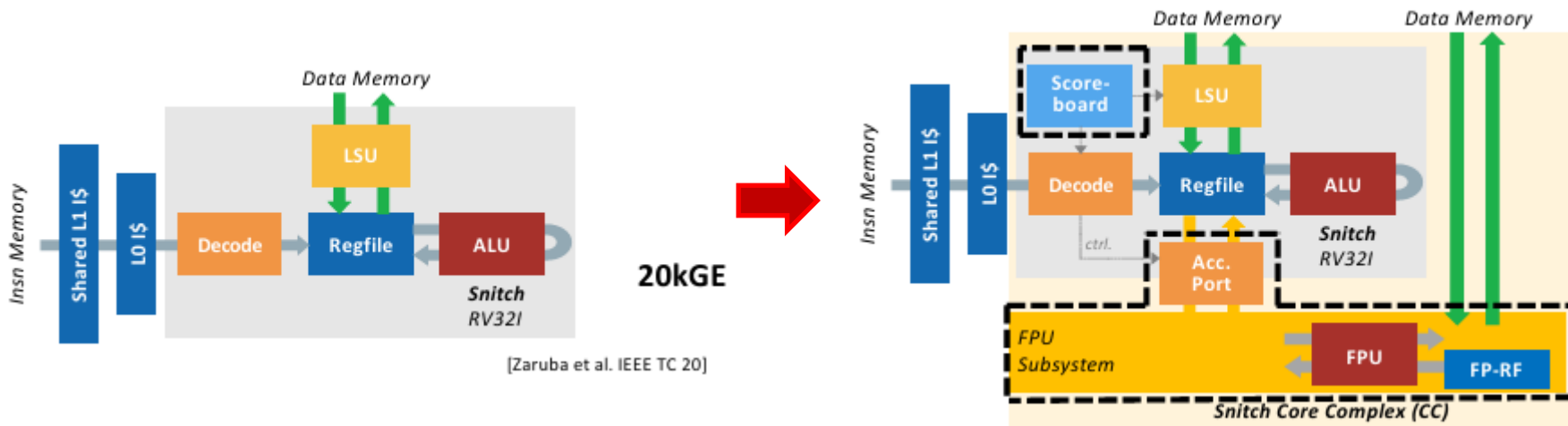
How to Determine Bit Width on DNN ?

- For accuracy, DNN operations decide bit width to achieve sufficient precision
- Which DNN operations affect the accuracy ?
 - **For inference:** weights, activations, and partial sums
 - **For training:** weights, activations, partial sums, gradients, and weight update
 - post-training quantization (PTQ)
 - A model compression technique that converts a pre-trained, full-precision model into a lower-precision model without needing to retrain or fine-tune it



Snitch core-complex

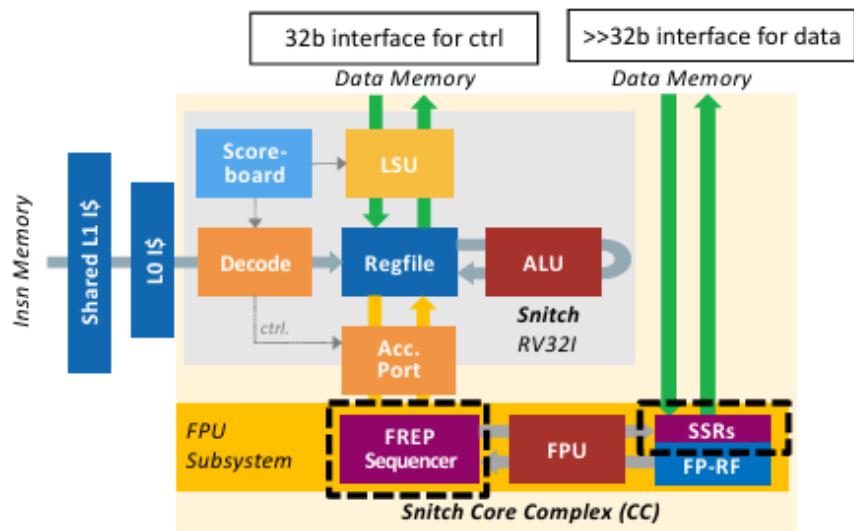
- Decouple PEs from control core through ISA-controlled Acceleration port
 - Multi-precision Floating-Point Units**, RVV, Tensor Units, AME extensions





Addressing the Von Neumann Bottleneck

- Accelerating hot loops with coarser-grained control/memory insns.
 - FREP -> hardware loop controller (repeats FPU ops autonomously)
 - SSR -> directly map memory streams to FP-RF with dense/sparse patterns (>> 32-bit bandwidth)



Streaming Semantic Registers (SSRs)

LD/ST elision

```
loop:  
fld r0, %[a]  
fld r1, %[b]  
fmadd r2, r0, r1
```



```
scfg 0, %[a], ldA  
scfg 1, %[b], ldB  
loop:  
fmadd r2, ssr0, ssr1
```

100% FPU utilization

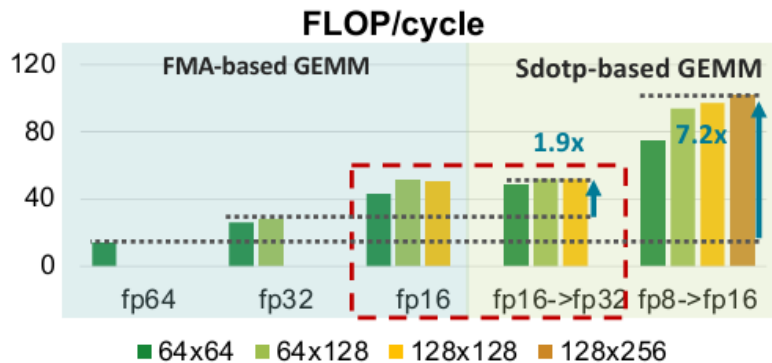
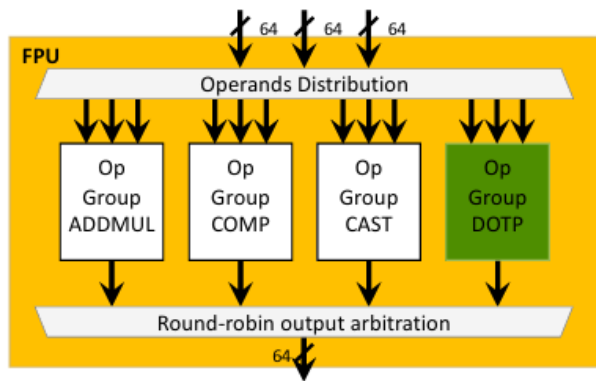
[Scheffler et al. IEEE JSSCC 25]

[Zaruba et al. IEEE TC 20]



Multi-precision FPU

- Support for FP64, FP32 + AI formats FP16 (BF16), FP8 (BF8) with SIMD
- Expanding DOTP unit (27% area overhead)
 - Minimize precision loss in accumulators (compared to FMA)
 - Dedicated OpGroup: two 16-to-32 (2 dotp/cyc) and four 8-to-16 units (4 dotp/cycle)



Results on a 8-core snitch-based cluster

[Bertaccini et al. IEEE TETO 20]



Low-Precision to the Rescue: Microscaling (MX)

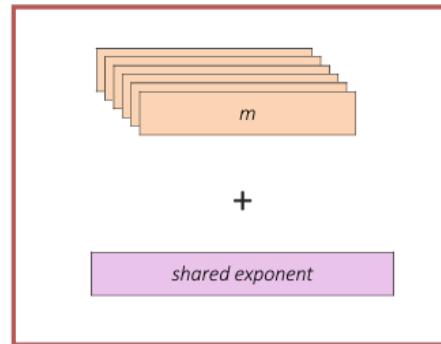
- Low-bit formats (FP8) reduce mem BW and compute cost
- But significant trade-off between range and precision
 - Smaller exponent -> **narrower dynamic range**
 - Smaller mantissa -> **reduced precision**
 - Outliers force a shared per-tensor scale -> all non-outliers may underflow/overflow and be represented as 0/max value, losing info



Low-Precision to the Rescue: Microscaling (MX)

- **Microscaling (MX) data formats**

- Offers finer-grained scaling an exponent over a block of 32 low-precision FP elements
- Dot-product computation by expanding FP formats in fixed-point -> accumulate “precisely” -> scale the block with shared exponent
- Enables low-bit formats while preserving accuracy
- **Inefficient if not supported in HW/ISA**

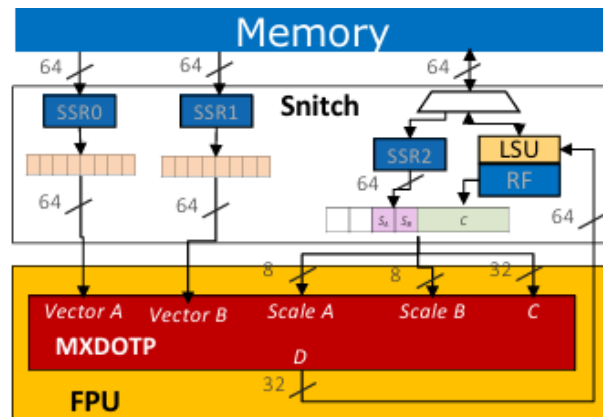
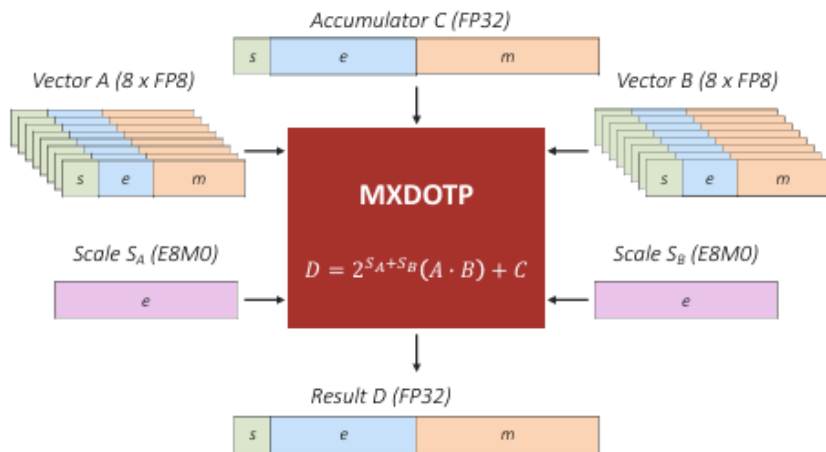


Block FP



Microscaling (MX) Formats for Transformers

- Scaled 8-way dotp (MXDOTP) insn for MXFP8 (E5M2, E4M3), MXINT8
 - Accumulation over a large **kulish** accumulator (e.g., 96b)
 - **Exactly** represent the sum of all partial products -> avoid swamping and rounding errors
 - Perform final block scaling with shared exponents





MXDOTP on MatMuls

Baseline-FP32

MatMul kernel with
FP32 inputs/outputs.

```
vfmac.s %[c], ft0, ft1
```

~ 0.5 instructions/MAC

Baseline-SW

MX MatMul kernel with
SW emulation (FP32 output).

```
lbu t1, 0(%[Sa])  
lbu t2, 0(%[Sb])  
add t3, t1, t2  
addi t3, t3, -127  
flh ft0, 0(%[A])  
vfcvt.s.b ft0, ft0  
flh ft4, 0(%[B])  
vfcvt.s.b ft4, ft4  
vfmac.s ft8, ft0, ft1  
slli t3, t3, 23  
fmv.w.x ft0, t3  
fmadd.s fa1, fa2, ft0, fa1
```

~ 3.9 instructions/MAC

MXDOTP ISA

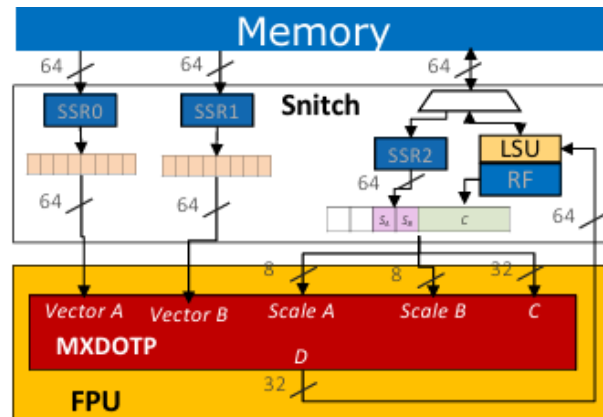
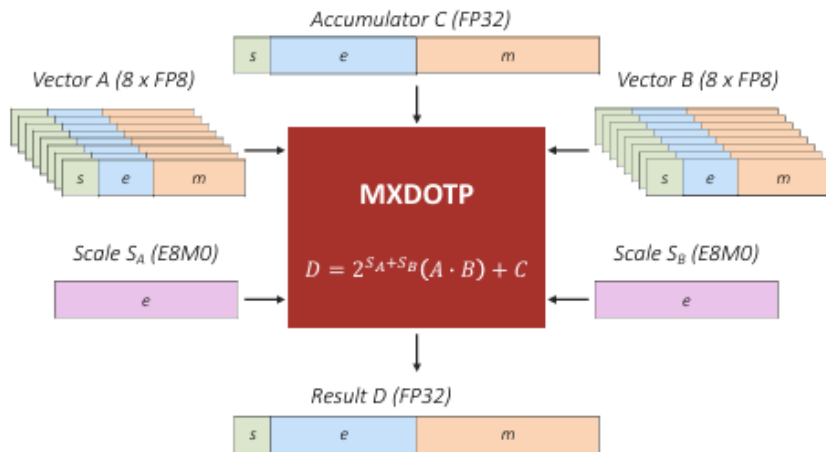
MX MatMul kernel using
MXDOTP instruction (FP32 out).

```
mxdotp %[c], ft0, ft1, ft2, sel
```

~ 0.16 instructions/MAC

Snitch-based Compute Cluster

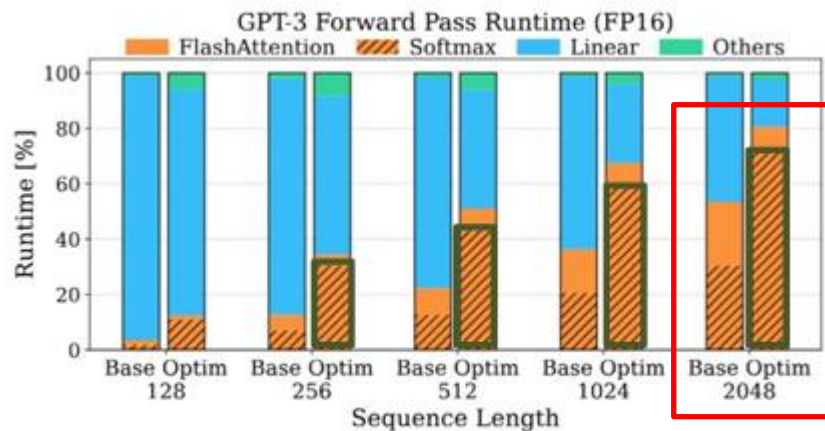
- 8 compute cores: 64b FPU + SSR + FREP
- 9th DMA core: for large (512-b AXI) cluster-to-cluster/memory 1D/2D transfers in HW (3D or more in SW), latency-tolerant blk transfers
- 128kb multi-banked scratchpad memory with wide DMA ports (SuperBanks)





Amdahl's Strikes Again: FlashAttention

- FlashAttention-2 workload
 - GEMM dominated, Softmax ~10%
- Head-level tiling across the multi-core MXU clusters
 - Each cluster computes Q/K/V tiles within assigned head
- As GEMM is optimized, bottleneck moves to non-linear functions
 - Softmax: dominated by floating point EXP operations





VFEXP: Lightweight FP EXP Acceleration

- How to accelerate Softmax and other exp-based non-linear functions?
 - Dedicated hardware -> high cost/utilization (Softmax only ~10% of the workload)
 - Accelerate EXP in the ISA + SW optimizations -> flexible, low-cost
- **EXP acceleration: Schraudolph's method + mantissa polynomial approximation**
 - **Faster** than CORDIC, **more flexible** than pure polynom. approx., **lower-cost** than LUTs
 - Implementation in **BF16** (extendable to other formats)



VFEXP: Lightweight FP EXP Acceleration

- **EXP acceleration: Schraudolph's method + mantissa polynomial approximation**
 - **Faster** than CORDIC, **more flexible** than pure polynom. approx., **lower-cost** than LUTs
 - Implementation in **BF16** (extendable to other formats)
 - EXP relative accuracy 99.86% (37x higher than schraudolph's)

Schraudolph's method

$$\text{exps}(x) \approx 2^{\text{int}(x)} \cdot (1 + \text{frac}(x))$$

+ mantissa correction

$$\text{exp}(x) \approx 2^{\text{int}(x)} \cdot (1 + \underline{P(\text{frac}(x))})$$

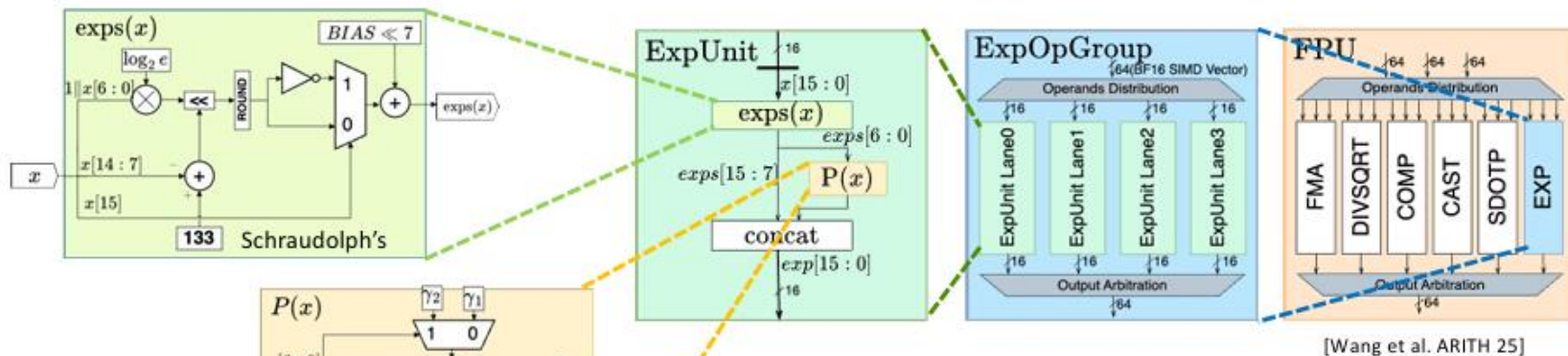
$$\underline{P(x)} = \begin{cases} \alpha x (x + \gamma_1), & x \in [0, 0.5) \\ \text{not } (\beta \text{ not } (x) \cdot (x + \gamma_2)), & x \in [0.5, 1) \end{cases}$$

[Belano et al. IEEE JETCAS 25]



VFEXP: Lightweight FP EXP Acceleration

- ExoOpGroup of 4 ExpUnit **BF16** lanes, 2 cycle of latency
- VFEXP (VEXP) instruction operating on **BF16** SIMD vectors (scalar BF16)
- 2.5% area overhead w.r.t. FPU (8kGE) – 537x less en/op wrt glibc's exp



Energy/Operation

Glibc's exp	VFEXP
3433 pJ/op	6.39 pJ/op

537x





Softmax Acceleration through ISA

FREP + SSR + glibc exp

```
// Find MAX
ssr config
frep.o %[n_frep], 1
vfmmax.h ft3,ft3,ft1

// EXP w/ GLIBC
for (int i = 0; i < N; i++) {
    y[i] = exp(x[i]- max_val)
    sum += y[i];}

// NORMALIZE
ssr config
frep.o %[n_frep], 1
vfdiv.h ft2,%[sum],ft1
```

350 Cycle/N

FREP + SSR + VFEXP

```
// Find MAX
ssr config
frep.o %[n_frep], 1, 0, 0
vfmmax.h ft3,ft3,ft0

// EXP w/ VFEXP
ssr config
frep.o %[n_frep], 4
vfsub.ah ft3, ft1, %[max]
vfexp.ah ft2, ft3
vfadd.ah ft4, ft2

// NORMALIZE
ssr config
frep.o %[n_frep], 1, 0, 0
vfmul.h ft4,%[1/sum],ft0
```

4 Cycle/N

..+ Loop Unrolling

```
// Find MAX
ssr config
frep.o %[n_frep],16
vfmmax.h ft3,ft3,ft0
vfmmax.h ft4,ft4,ft0
vfmmax.h ft5,ft5,ft0

// EXP w/ VFEXP
ssr config
frep.o %[n_frep], 16
vfsub.ah ft3, ft1, %[max]
vfsub.ah ft4, ft1, %[max]
vfexp.ah ft3, ft3
vfexp.ah ft4, ft4
vfadd.ah ft3, ft3
vfadd.ah ft4, ft4

// NORMALIZE
ssr config
frep.o %[n_frep], 16
vfmul.h ft4,%[1/sum],ft0
vfmul.h ft4,%[1/sum],ft0
vfmul.h ft4,%[1/sum],ft0
.....
```

2.125 Cycle/N

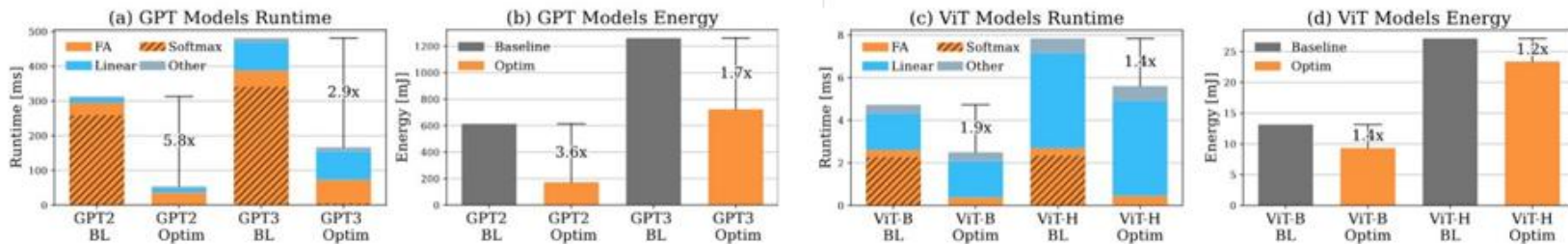
- ❑ 165x higher performance than baseline on Softmax
- ❑ 74.3x better energy efficiency than baseline on Softmax
- ❑ Similar optimizations apply for GeLu and other exp-based non-linear fx



VFEXP: Lightweight FP EXP Acceleration

- Very limited area/power cost (<1% overhead at core level)
- > 10-100x improved performance and energy efficiency on Softmax
- Negligible accuracy loss on end-to-end execution of GPT-x/ViT models
- Up to 5.8x latency and 3.6x energy reduction on end-to-end workloads

End-to-End Non-autoregressive Inference



End-to-End Accuracy

Model	Dataset	Metric	FP32	BF16	BF16 EXP
GPT-2	WikiText	Perplexity (↓)	37.4	37.8	37.8
	ArcEasy	Accuracy (↑)	43.8	42.9	43.7
ViT-B	ImageNet	Accuracy (↑)	80.3	80.3	80.3
	CIFAR-10	Accuracy (↑)	98.5	98.5	98.5

[Wang et al. ARITH 25]



Takeaway Questions

- What are advantages to use BF16 instead of FP16 ?
 - (A) Fast conversion from FP32
 - (B) Get more precise value
 - (C) Represent few different values
- What are benefits to use lower precision data type on neural network ?
 - (A) Reduce the latency of DNN models
 - (B) Save the memory space
 - (C) Lower the power consumption of the accelerator



K-Means-based Weight Quantization

- **Storage**
 - Integer Weights; Floating-Point Codebook
- **Computation**
 - Floating-Point Arithmetic

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

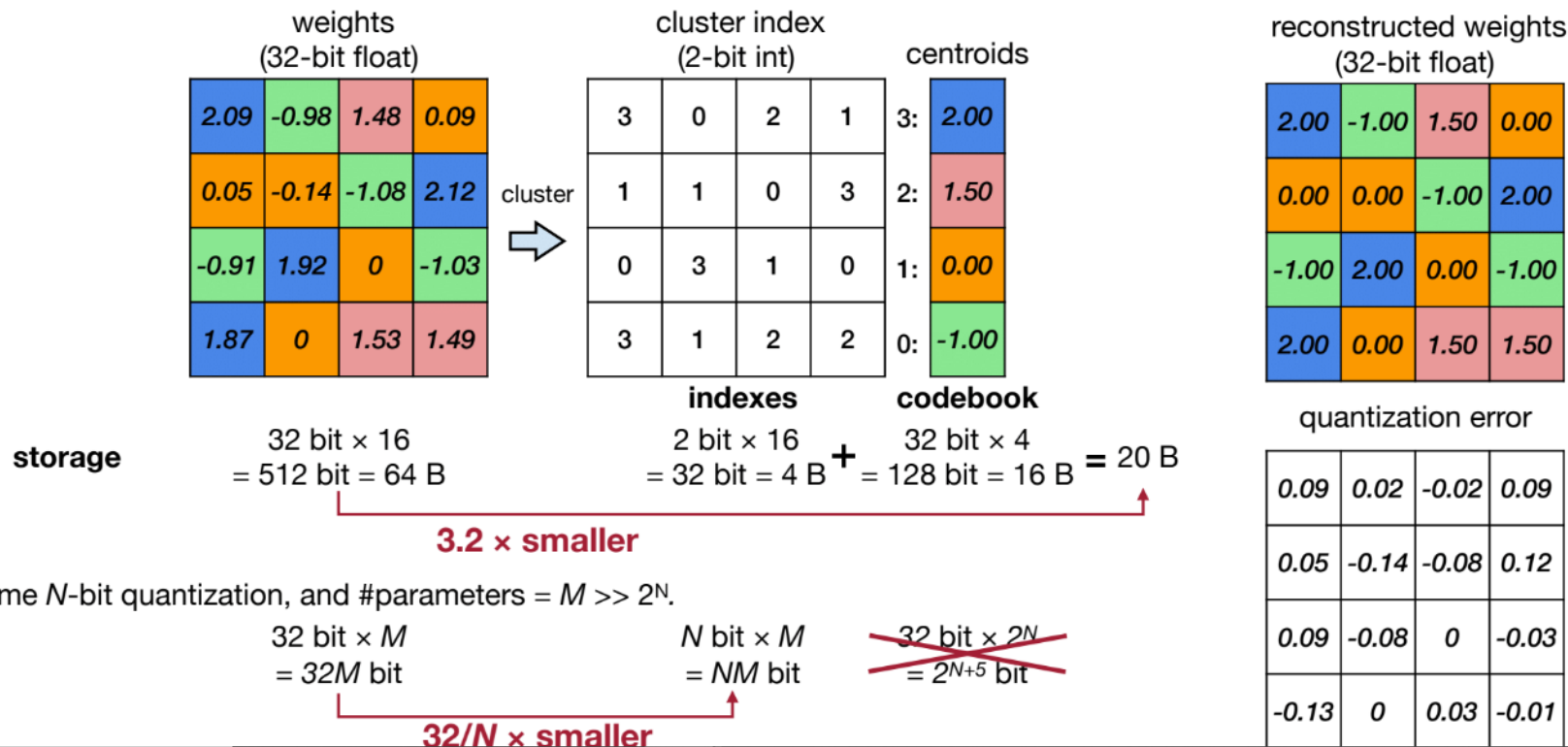
~~2.09, 2.12, 1.92, 1.87~~



2.0



K-Means-based Weight Quantization



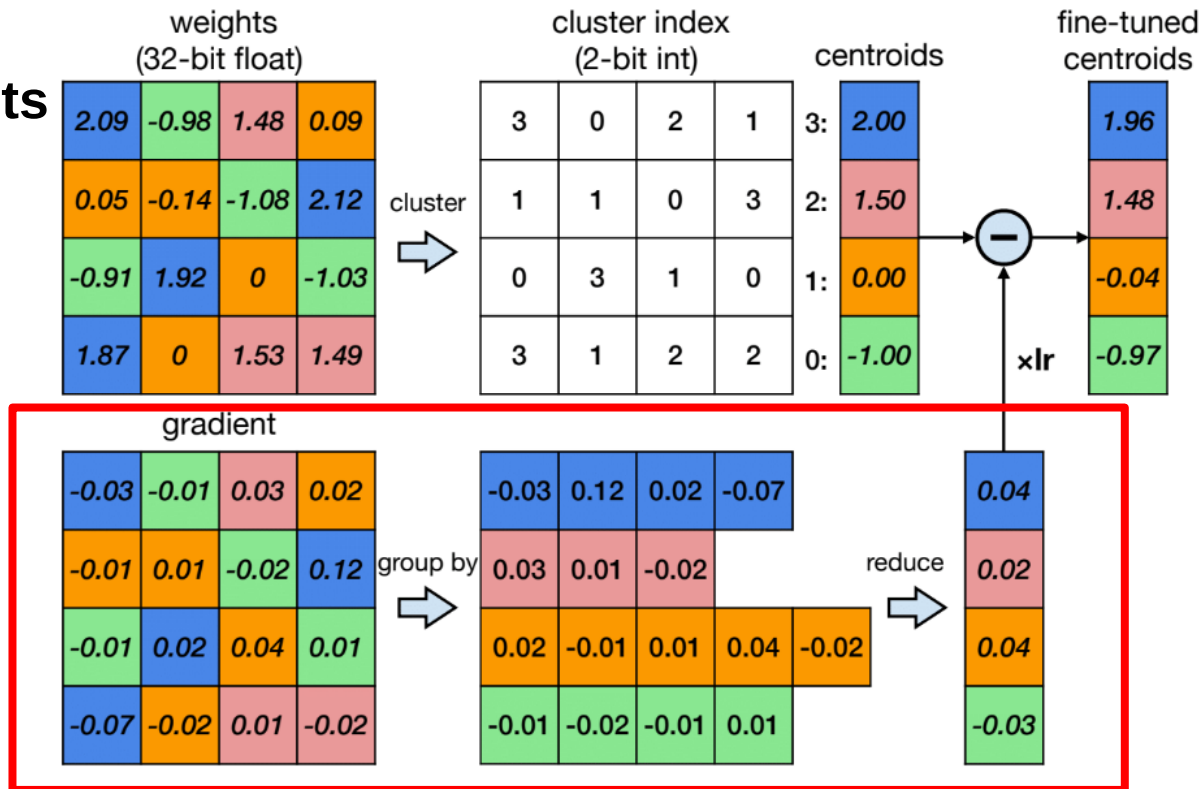


K-Means-based Weight Quantization

- **Fine-tuning**

Quantized Weights

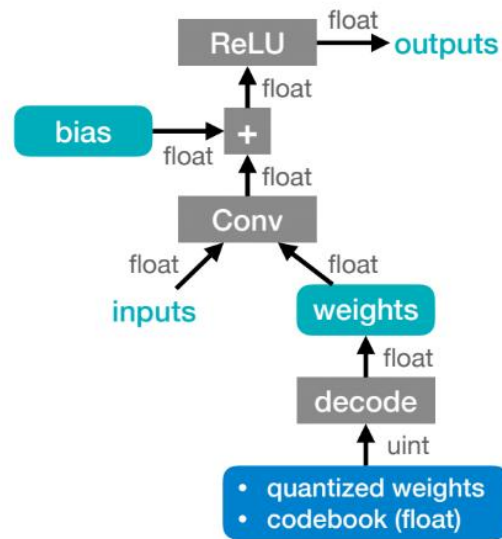
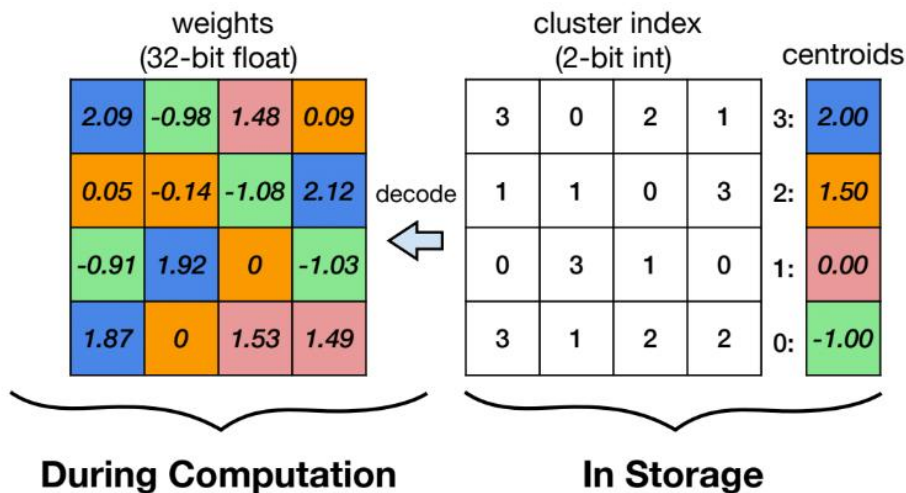
- Reduce the quantization error





K-Means-based Weight Quantization

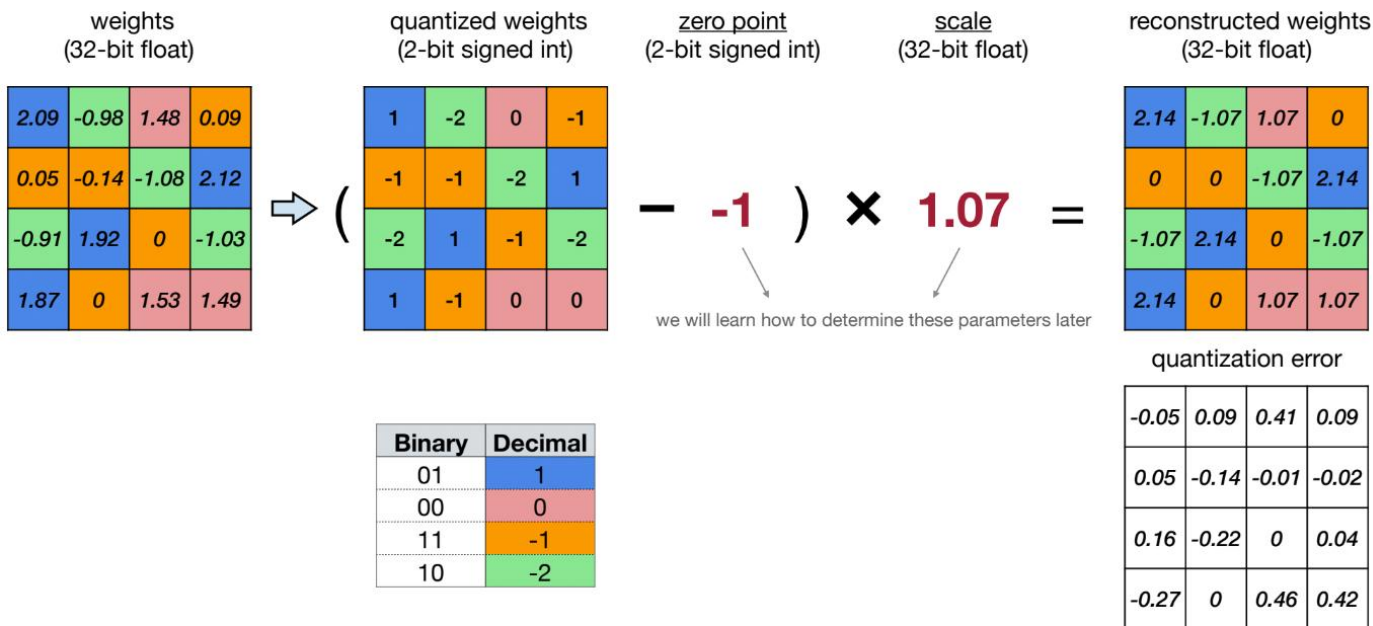
- Weights are decompressed using a lookup table during runtime inference
- Only saves storage cost of a neural network model
- All the computation and memory access are still floating-point





What is Linear Quantization ?

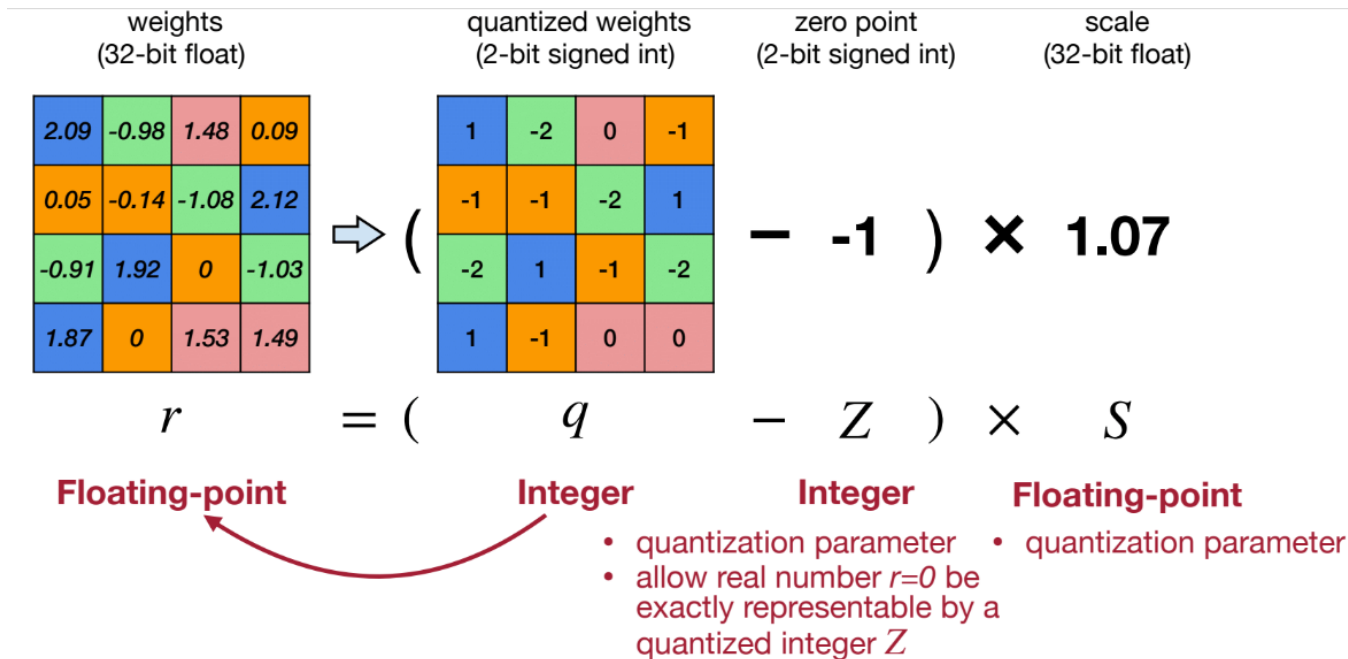
- An affine mapping of integers to real numbers
- **Storage:** Integer Weights; **Computation:** Integer Arithmetic





Linear Quantization

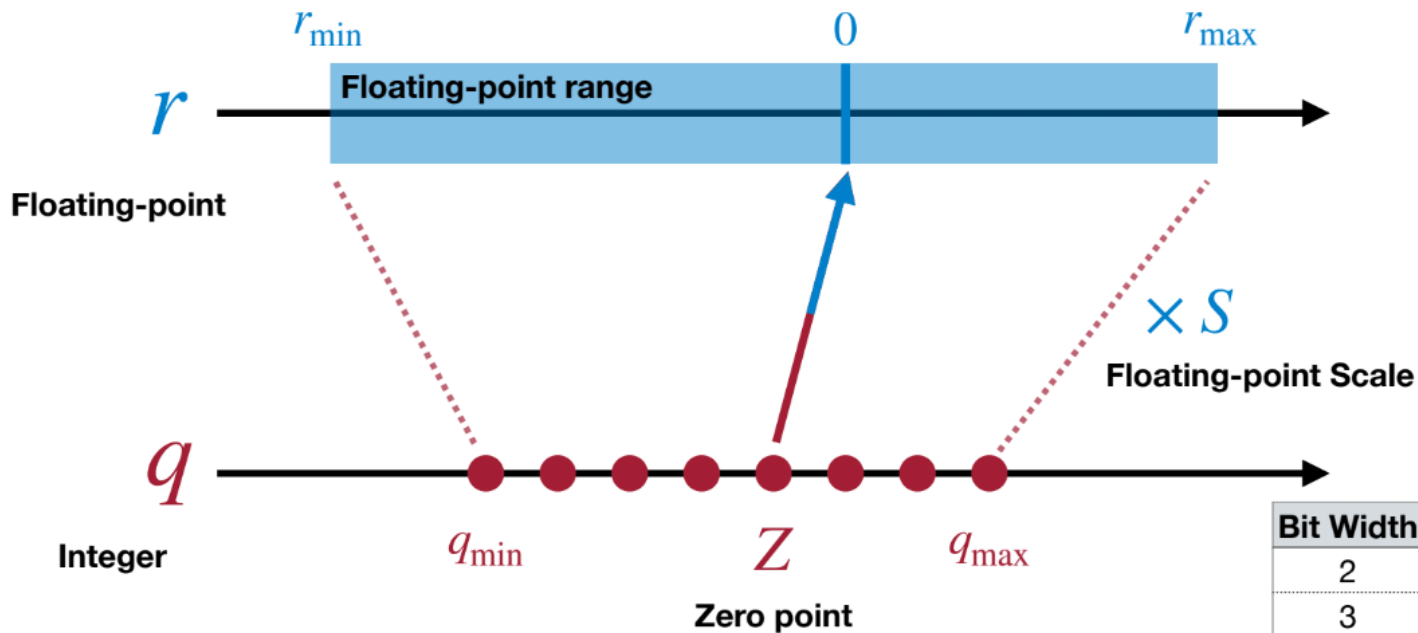
- An affine mapping of integers to real numbers ($r = S(q - Z)$)





Linear Quantization

- An affine mapping of integers to real numbers ($r = S(q - Z)$)

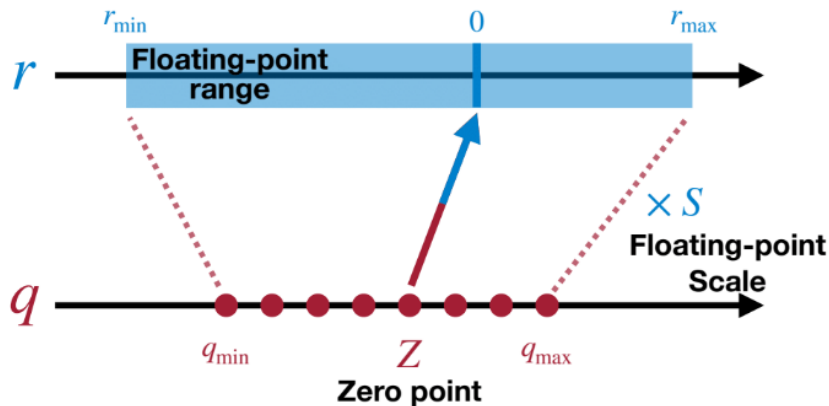


Bit Width	$q_{\min}$	$q_{\max}$
2	-2	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$



Scale of Linear Quantization

- An affine mapping of integers to real numbers ($r = S(q - Z)$)



$$\begin{aligned} r_{\max} &= S(q_{\max} - Z) \\ r_{\min} &= S(q_{\min} - Z) \end{aligned} \quad \ominus$$

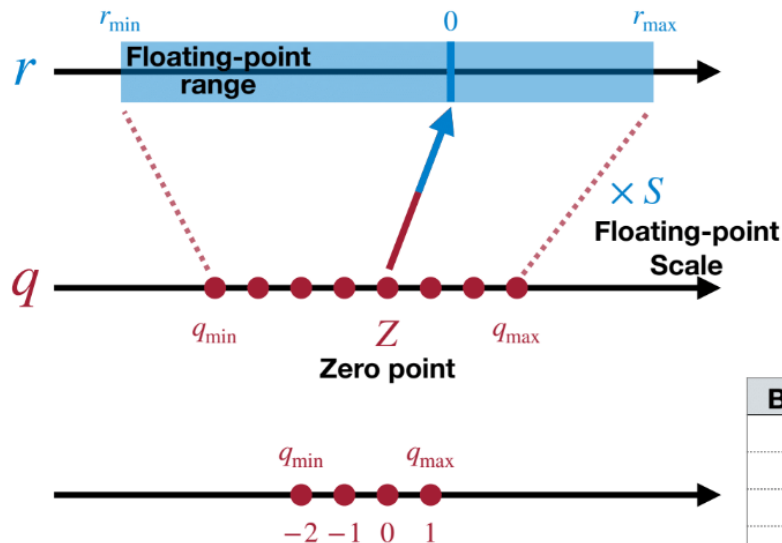
$$\downarrow$$
$$r_{\max} - r_{\min} = S(q_{\max} - q_{\min})$$

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



Scale of Linear Quantization

- An affine mapping of integers to real numbers ($r = S(q - Z)$)



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

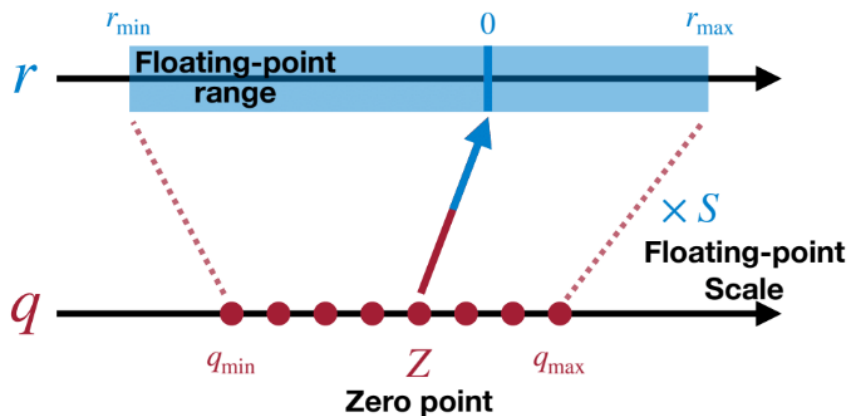
$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$$
$$= \frac{2.12 - (-1.08)}{1 - (-2)}$$
$$= 1.07$$

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



Zero Point of Linear Quantization

- An affine mapping of integers to real numbers ($r = S(q - Z)$)



$$r_{\min} = S(q_{\min} - Z)$$

↓

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

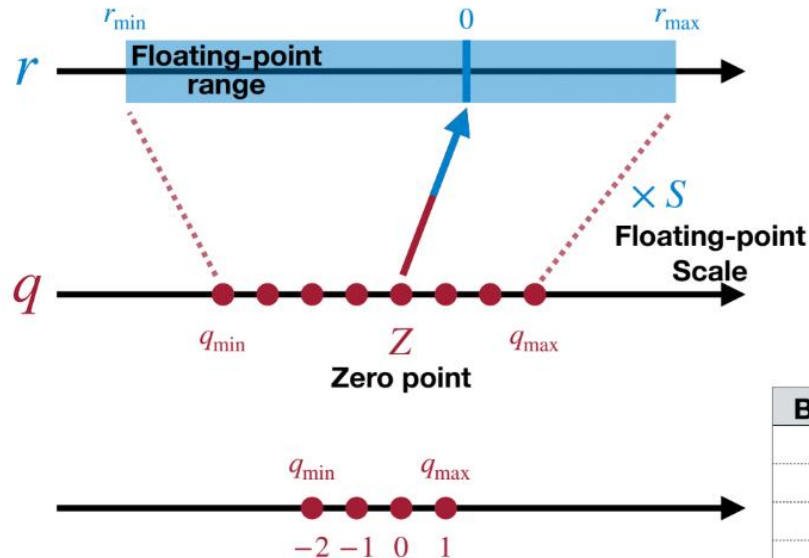
↓

$$Z = \text{round} \left(q_{\min} - \frac{r_{\min}}{S} \right)$$



Zero Point of Linear Quantization

- An affine mapping of integers to real numbers ($r = S(q - Z)$)



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

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

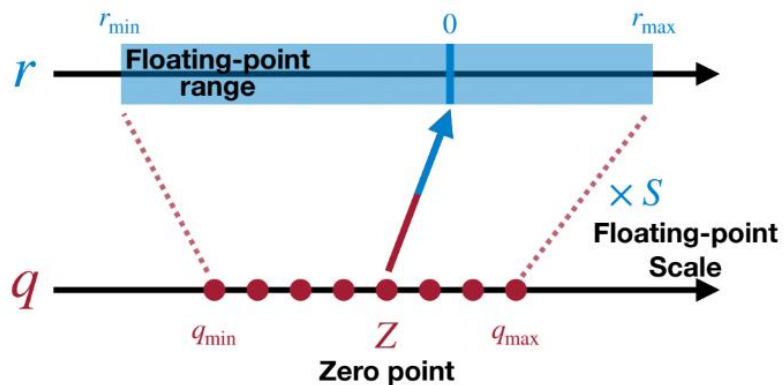
$$\begin{aligned} &= \text{round}\left(-2 - \frac{-1.08}{1.07}\right) \\ &= -1 \end{aligned}$$

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



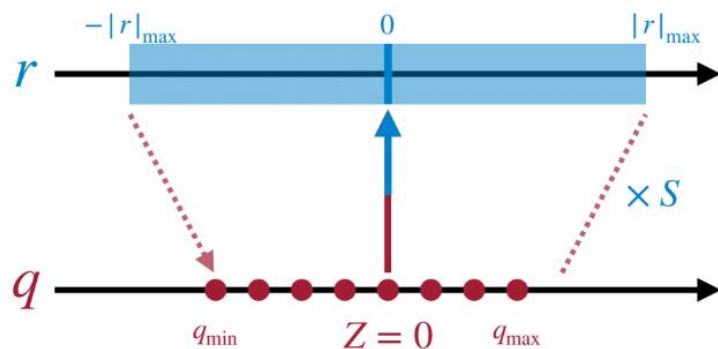
Asymmetric Linear Quantization

- Full range mode



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

Bit Width	$q_{\min}$	$q_{\max}$
2	-2	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$



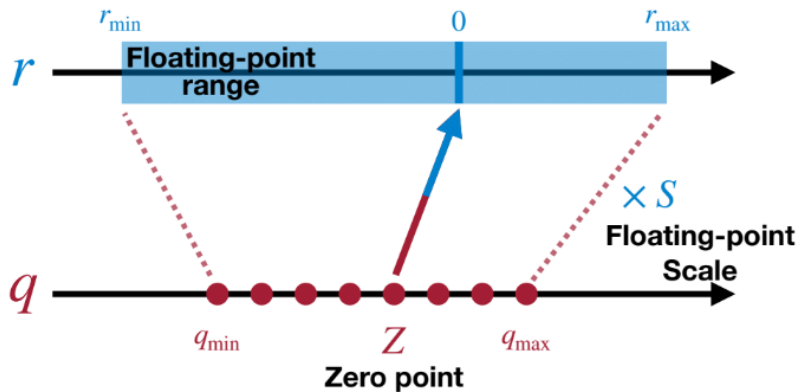
$$S = \frac{r_{\min}}{q_{\min} - Z} = \frac{-|r|_{\max}}{q_{\min}} = \frac{|r|_{\max}}{2^{N-1}}$$

- use full range of quantized integers
- example: PyTorch's native quantization, ONNX



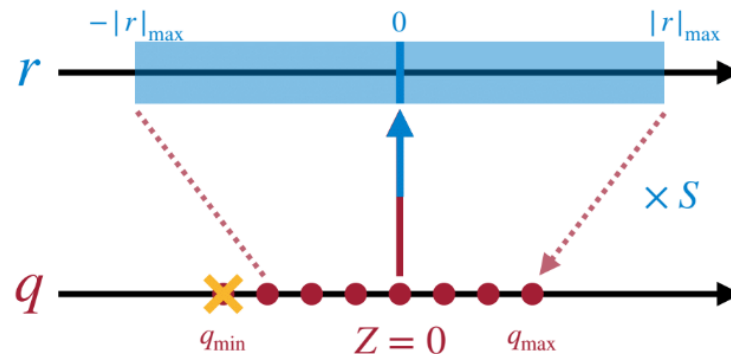
Symmetric Linear Quantization

- Restricted range mode



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

Bit Width	$q_{\min}$	$q_{\max}$
2	-2	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$

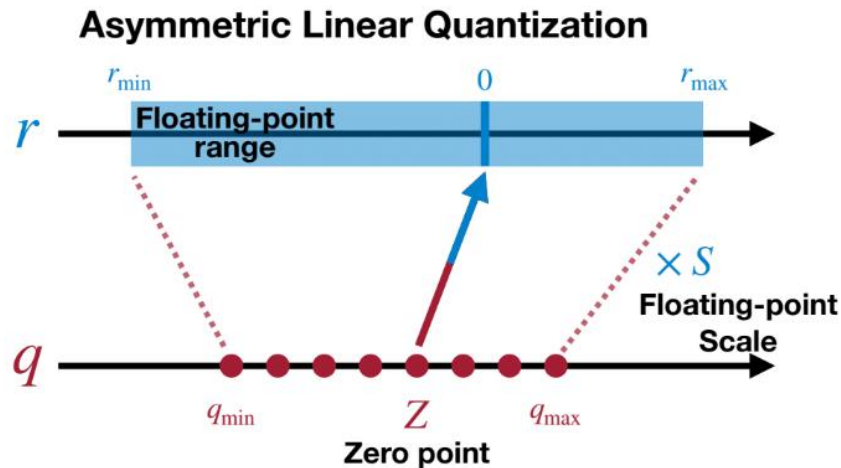


$$S = \frac{r_{\max}}{q_{\max} - Z} = \frac{|r|_{\max}}{q_{\max}} = \frac{|r|_{\max}}{2^{N-1} - 1}$$

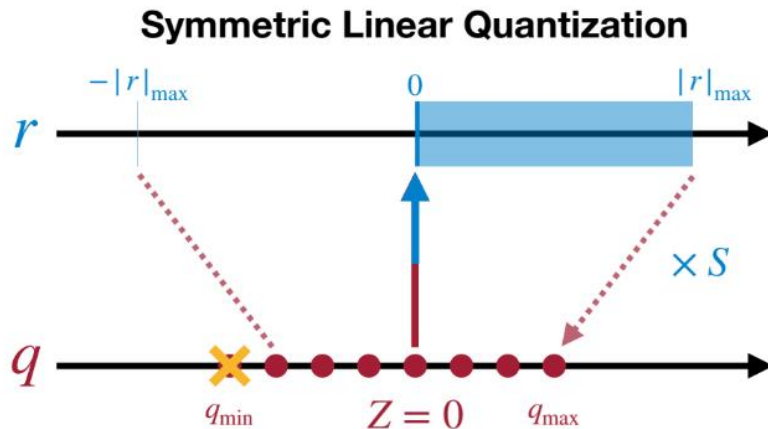
- example: TensorFlow, NVIDIA TensorRT, Intel DNNL



Asymmetric vs. Symmetric



- The quantized range is fully used.
- The implementation is more complex, and zero points require additional logic in hardware.



- The quantized range will be wasted for biased float range.
 - Activation tensor is non-negative after ReLU, and thus symmetric quantization will lose 1 bit effectively.
- The implementation is much simpler.



Binary/Ternary Quantization

- Could we push the quantization precision to 1 bit?

$$y_i = \sum_j W_{ij} \cdot x_j$$
$$= 8 \times 5 + (-3) \times 2 + 5 \times 0 + (-1) \times 1$$

$$\begin{bmatrix} \\ y_i \\ \\ \end{bmatrix} = \begin{bmatrix} & & & \\ & W_{ij} & & \\ & & & \\ & & & \end{bmatrix} \times \begin{bmatrix} \\ x_j \\ \\ \end{bmatrix}$$

$$\begin{bmatrix} \\ \\ \\ \end{bmatrix} = \begin{bmatrix} & & & \\ 8 & -3 & 5 & -1 \\ & & & \\ & & & \end{bmatrix} \times \begin{bmatrix} 5 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$

input	weight	operations	memory	computation
$\mathbb{R}$	$\mathbb{R}$	$+ \times$	$1 \times$	$1 \times$



Binary/Ternary Quantization

- If weights are quantized to +1 and -1

$$y_i = \sum_j W_{ij} \cdot x_j$$
$$= 5 - 2 + 0 - 1$$

input	weight	operations	memory	computation
$\mathbb{R}$	$\mathbb{R}$	$+ \times$	$1\times$	$1\times$
$\mathbb{R}$	$\mathbb{B}$	$+ -$	$\sim 32\times$ less	$\sim 2\times$ less

$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 & 2 & 0 & 1 \\ 8 & -3 & 5 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 5 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 5 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$



Binarization

- **Deterministic Binarization**

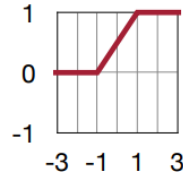
- directly computes the bit value based on a threshold, usually 0, resulting in a sign function.

$$q = \text{sign}(r) = \begin{cases} +1, & r \geq 0 \\ -1, & r < 0 \end{cases}$$

- **Stochastic Binarization**

- use global statistics or the value of input data to determine the probability of being -1 or +1
 - e.g., in Binary Connect (BC), probability is determined by hard sigmoid function $\sigma(r)$

$$q = \begin{cases} +1, & \text{with probability } p = \sigma(r) \\ -1, & \text{with probability } 1 - p \end{cases}, \quad \text{where } \sigma(r) = \min(\max(\frac{r+1}{2}, 0), 1)$$



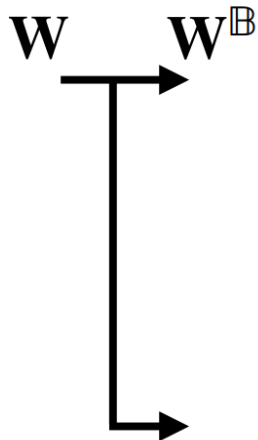
- harder to implement as it requires the hardware to generate random bits when quantizing.



Minimizing Quantization Error in Binarization

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



binary weights
(1-bit)

1	-1	1	1
1	-1	-1	1
-1	1	1	-1
1	1	1	1

1	-1	1	1
1	-1	-1	1
-1	1	1	-1
1	1	1	1

$$\mathbf{W}^{\mathbb{B}} = \text{sign}(\mathbf{W})$$

$$\alpha = \frac{1}{n} \|\mathbf{W}\|_1$$

$$\|\mathbf{W} - \mathbf{W}^{\mathbb{B}}\|_F^2 = 9.28$$

scale
(32-bit float)

$$\times 1.05 = \frac{1}{16} \|\mathbf{W}\|_1$$

$$\|\mathbf{W} - \alpha \mathbf{W}^{\mathbb{B}}\|_F^2 = 9.24$$

AlexNet-based Network	ImageNet Top-1 Accuracy Delta
BinaryConnect	-21.2%
Binary Weight Network (BWN)	0.2%



Binary Net

- **Binary Connect**

- Weights $\{-1, 1\}$ (Bipolar binary),
Activation 32-bit float
- Accuracy loss: 19 % on AlexNet

- **Binarized Neural Networks**

- Weights $\{-1, 1\}$, Activations $\{-1, 1\}$
- Both of operands are binary, the multiplication turns into an XNOR
- Accuracy loss: 29.8 % on AlexNet

XNOR

A	B	Out
0	0	1
1	0	0
0	1	0
1	1	1

Popcount (110010001) = 4

for each i in width:

$C += A[\text{row}][i] * B[i][\text{col}]$



for each i in width:

$C += \text{popcount}(\text{XNOR}(A[\text{row}][i], B[i][\text{col}]))$



Case Study: Binary Multiplication

- $A = 10010$, $B = 01111$ (0 is really -1 here)
- **Dot product:**
 - $A * B = (1 * -1) + (-1 * 1) + (-1 * 1) + (1 * 1) + (-1 * 1) = -3$
- $P = \text{XNOR}(A, B) = 00010$, $\text{popcount}(P) = 1$
- $\text{Result} = 2 * P - N$, where N is the total number of bits
- $2 * P - N = 2 * 1 - 5 = -3$



XNOR-Net

- If both activations and weights are binarized

$$\begin{aligned}y_i &= \sum_j W_{ij} \cdot x_j \\&= 1 \times 1 + (-1) \times 1 + 1 \times (-1) + (-1) \times 1 \\&= 1 + (-1) + (-1) + (-1) = -2\end{aligned}$$

$$\begin{bmatrix} \square \\ \blacksquare \\ \square \\ \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square & \square \\ \blacksquare & 8 & -3 & 5 & -1 \\ \square & \square & \square & \square \\ \square & \square & \square & \square \end{bmatrix} \times \begin{bmatrix} \blacksquare 5 \\ \blacksquare 2 \\ \blacksquare 0 \\ \blacksquare 1 \end{bmatrix}$$

$$\begin{bmatrix} \square \\ \blacksquare \\ \square \\ \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square & \square \\ \blacksquare & 1 & -1 & 1 & -1 \\ \square & \square & \square & \square \\ \square & \square & \square & \square \end{bmatrix} \times \begin{bmatrix} \blacksquare 1 \\ \blacksquare 1 \\ \blacksquare -1 \\ \blacksquare 1 \end{bmatrix}$$



XNOR-Net

- If both activations and weights are binarized

$$\begin{aligned} y_i &= \sum_j W_{ij} \cdot x_j \\ &= 1 \times 1 + (-1) \times 1 + 1 \times (-1) + (-1) \times 1 \\ &= 1 + (-1) + (-1) + (-1) = -2 \end{aligned}$$

W	X	Y=WX
1	1	1
1	-1	-1
-1	-1	1
-1	1	-1

b _w	b _x	XNOR(b _w , b _x)
1	1	1
1	0	0
0	0	1
0	1	0



XNOR-Net

- If both activations and weights are binarized

$$y_i = \sum_j W_{ij} \cdot x_j$$

$$\begin{aligned} &= 1 \times 1 + (-1) \times 1 + 1 \times (-1) + (-1) \times 1 \\ &= 1 + (-1) + (-1) + (-1) = -2 \end{aligned}$$

$$\begin{aligned} &= 1 \text{ xnor } 1 + 0 \text{ xnor } 1 + 1 \text{ xnor } 0 + 0 \text{ xnor } 1 \\ &= 1 + 0 + 0 + 0 = 1 \end{aligned}$$

↑ ? ↓

W	X	Y=WX
1	1	1
1	-1	-1
-1	-1	1
-1	1	-1

b _w	b _x	XNOR(b _w , b _x)
1	1	1
1	0	0
0	0	1
0	1	0



XNOR-Net

- If both activations and weights are binarized

$$y_i = \sum_j W_{ij} \cdot x_j$$

$$= 1 \times 1 + (-1) \times 1 + 1 \times (-1) + (-1) \times 1$$

$$= 1 + (-1) + (-1) + (-1) = -2$$

$$y_i = -n + 2 \cdot \sum_j W_{ij} \text{ xnor } x_j$$

$$= 1 \text{ xnor } 1 + 0 \text{ xnor } 1 + 1 \text{ xnor } 0 + 0 \text{ xnor } 1$$

$$= 1 + 0 + 0 + 0 = \boxed{1 \times 2 + (-4)} = -2$$

↑ +2
Assuming -1 -1 -1 -1 → -4

W	X	Y=WX
1	1	1
1	-1	-1
-1	-1	1
-1	1	-1

b _w	b _x	XNOR(b _w , b _x)
1	1	1
1	0	0
0	0	1
0	1	0



XNOR-Net

- If both activations and weights are binarized

$$y_i = -n + 2 \cdot \sum_j W_{ij} \text{ xnor } x_j \quad \rightarrow \quad y_i = -n + \text{popcount}(W_i \text{ xnor } x) \ll 1$$
$$= -4 + 2 \times (1 \text{ xnor } 1 + 0 \text{ xnor } 1 + 1 \text{ xnor } 0 + 0 \text{ xnor } 1)$$
$$= -4 + 2 \times (1 + 0 + 0 + 0) = -2$$

→ popcount: return the number of 1

W	X	Y=WX
1	1	1
1	-1	-1
-1	-1	1
-1	1	-1

b _w	b _x	XNOR(b _w , b _x)
1	1	1
1	0	0
0	0	1
0	1	0



XNOR-Net

- If both activations and weights are binarized

$$y_i = -n + \text{popcount}(W_i \text{ xnor } x) \ll 1$$

$$= -4 + \text{popcount}(1010 \text{ xnor } 1101) \ll 1$$

$$= -4 + \text{popcount}(1000) \ll 1 = -4 + 2 = -2$$

input	weight	operations	memory	computation
$\mathbb{R}$	$\mathbb{R}$	$+ \times$	$1\times$	$1\times$
$\mathbb{R}$	$\mathbb{B}$	$+ -$	$\sim 32\times$ less	$\sim 2\times$ less
$\mathbb{B}$	$\mathbb{B}$	xnor, popcount	$\sim 32\times$ less	$\sim 58\times$ less

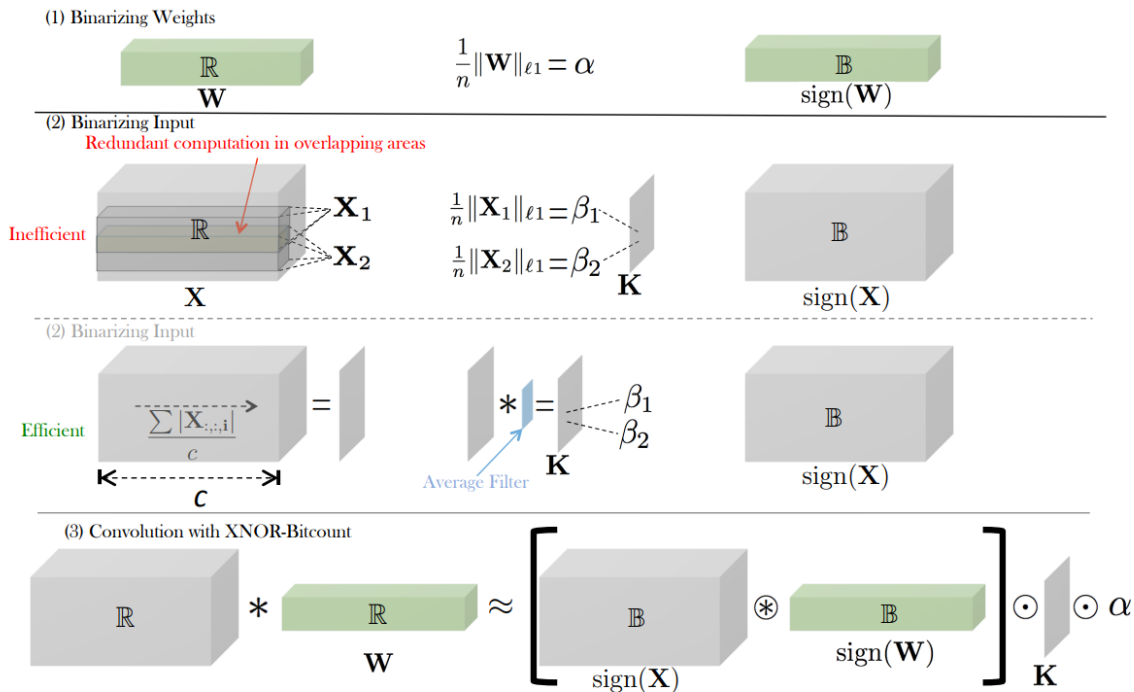
$$\begin{bmatrix} 0 \\ 8 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 8 & -3 & 5 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 5 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$



XNOR-Net

- Minimizing quantization error in binarization





XNOR-Net

Neural Network	Quantization	Bit-Width		ImageNet Top-1 Accuracy Delta
		W	A	
AlexNet	BWN	1	32	0.2%
	BNN	1	1	-28.7%
	XNOR-Net	1	1	-12.4%
GoogleNet	BWN	1	32	-5.80%
	BNN	1	1	-24.20%
ResNet-18	BWN	1	32	-8.5%
	XNOR-Net	1	1	-18.1%

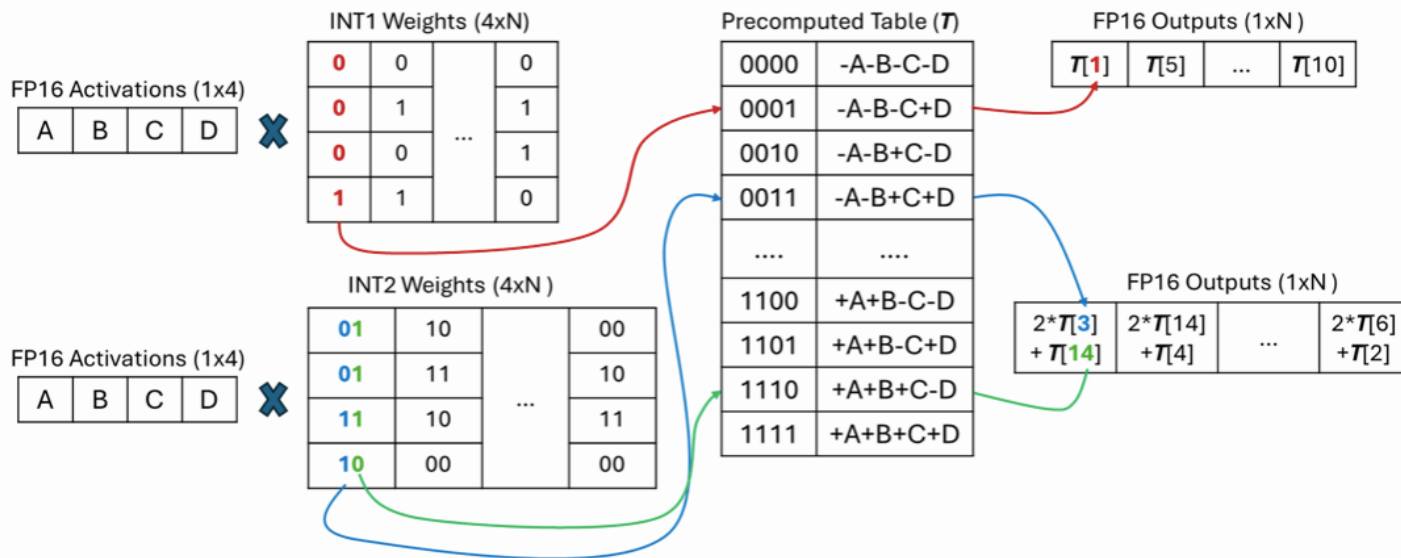
- * BWN: Binary Weight Network with scale for weight binarization
- * BNN: Binarized Neural Network without scale factors
- * XNOR-Net: scale factors for both activation and weight binarization



BitNet

- FP16 activation and 1.58 bit weights – Transformer-based model
- Lookup table (LUT) calculations

<https://www.arxiv.org/pdf/2407.00088>





What do we Learn from Quantization?

- **Quantization** can improve DNN computational throughput while maintaining accuracy
- Layers on DNN models can be offered with **different bit widths**
- Varying bit width requires **the support of the hardware**
- **No systematic approach** to figure out the proper bit width in layers of DNN models
- What else ?



Takeaway Questions

- What are purposes of data quantization ?
 - (A) Constrain the value of inputs to a set of discrete values
 - (B) Create more values
 - (C) Improve the degree of parallelism on DNN training
- Why training requires large bit width ?
 - (A) The training needs to compute more data
 - (B) Avoid the value underflow and overflow
 - (C) Gradient and weight update have a larger range