Paper Seminar Presentation #2

BitMoD: Bit-serial Mixture-of-Datatype LLM Acceleration, HPCA 2025

2022142255 SeokChan Jeong

Table of Content

Motivation and Background

Why Weight Quantization for LLMs?

LLM inference stores two major tensors

Weight

- Model parameters
- Huge memory footprint
- Fetched repeatedly during inference

Activation

- Intermediate data
- Depends on input and batch size
- Reuse is limited at small batch

Table 1: The memory consumption of model weights and KV cache for three different LLMs at batch size 128 and sequence length 2048 shows that the KV cache dominates the memory consumption.

Model	# of Layer	Hidden Size	Weights (GB)	KV cache (GB)
OPT-175B	96	12288	325	1152
LLaMA-65B	80	8192	130	640
BLOOM	70	14336	352	950

Zichang Liu et al., *"Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time"*, **NeurIPS 2023**.

Batch size = 1 or small batch

One request / one token step

Activation reuse is limited

Large weight matrices are repeatedly fetched

Weight Memory

Matrix Multiply

Output Token

FP16 → Low-bit Weight Quantization

Main operation: represent the same weight using fewer bits

FP16 Weight

16 bits / weight

16 bit → **4 bit**

≈ 1/4 storage & memory traffic

Low-bit Weight

4 bits / weight

3 bits / weight

Examples

- FP16 → INT4 or FP4: 16-bit weight becomes 4-bit weight.
- FP16 → FP3: even smaller memory, but accuracy becomes harder to preserve.
- The challenge is to reduce memory while keeping model quality.

QAT vs PTQ: Two Quantization Paths

Quantization method determines whether we retrain the model or only convert a trained model.

QAT	Quantization-Aware Training	PTQ	Post-Training Quantization

- The model is trained while quantization error is simulated.
- Can achieve higher accuracy because training adapts to quantization noise.
- But retraining a large LLM is extremely expensive.

Retraining: Required

- The model is quantized after it has already been trained.
- No full retraining is required, so it is practical for LLM deployment.
- Accuracy can drop more than QAT, so better data types are needed.

Retraining: Not required

For LLMs **PTQ is generally preferred because retraining full LLMs is too costly.**

Why Weight-only PTQ is Widely Used

Most LLM quantization work focuses on weights, while keeping activations in FP16.

Operand	Treatment	Reason
Weight	Quantized to low precision	Large memory saving
Activation	Kept as FP16	Preserves accuracy and avoids activation quantization difficulty
Result	Low-precision Weight × FP16 Activation	Good memory-quality trade-off, but creates a compute mismatch

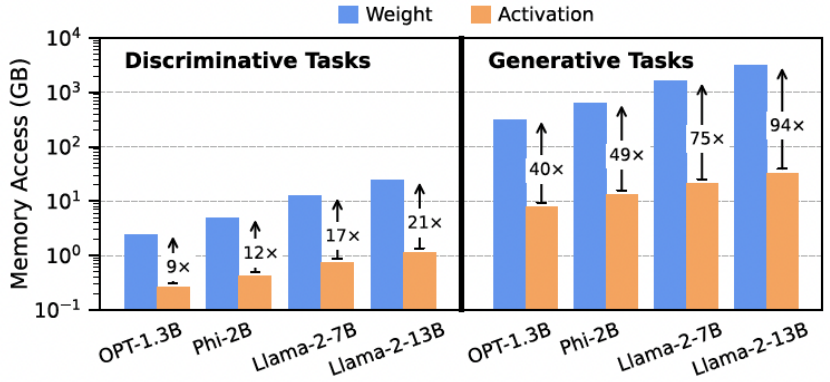
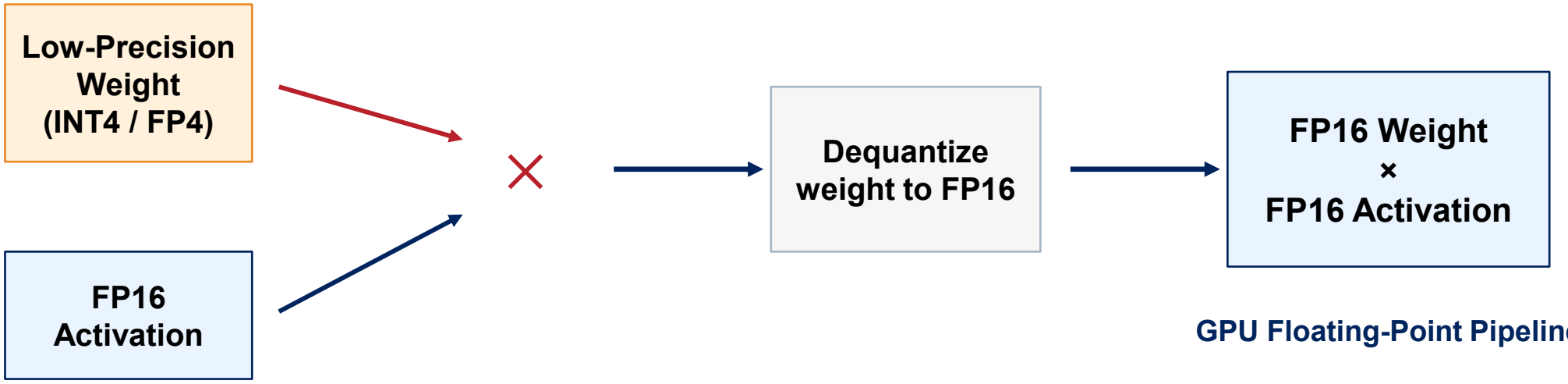


Fig. 1: Total memory access of weights and activations on discriminative tasks (with 256 input tokens and 1 output token) and generative tasks (with 256 input tokens and 256 generated tokens). Note the log scale on the y-axis. Note that the gap between weight and activation memory accesses increases for generative tasks at batch size 1 despite a much larger KV-cache than discriminative tasks. While prior work [44] has correctly reported a memory bottleneck caused by the KV-cache, this only occurs for 175B+ parameter models with a high batch size (e.g., 512) and a context lengths exceeding 512 tokens. This scenario is less relevant to our focus on low-batch edge LLM inference where the weights indeed dominate the total memory accesses.

Low-bit Weight × **FP16 Activation** → **Memory saving is achieved, but hardware must still compute mixed precision efficiently.**

Limitation of Weight-only PTQ on GPUs

Problem: low-bit storage does not automatically mean low-bit compute



Therefore

- Weight-only PTQ saves memory, but GPUs often restore low-bit weights to FP16 before computation.
- The advantage of low-precision weight is not fully translated into compute efficiency.
- BitMoD targets this gap with hardware that directly handles low-precision weight × FP16 activation.

Existing Approaches Before BitMoD

After weight-only PTQ, prior work tried to avoid the “low-bit storage but FP16 compute” problem in different ways.

FIGNA

Integer Weight × Floating-point Activation

Goal: compute quantized weights directly, instead of restoring all weights to FP16.

Microscaling

Low-precision group + shared exponent

Goal: reduce exponent overhead by sharing scale-like information across several values.

ANT

Adaptive Numeric Type

Goal: choose a data type that better matches tensor distribution and reduces quantization error.

OliVe

Outlier-aware value encoding

Goal: protect very large outliers by sacrificing small “victim” values.

Common goal: keep the memory benefit of low-bit weights while making the hardware compute path efficient

Compare with BitMoD

Existing approaches solve part of the problem, but not all requirements at the same time.

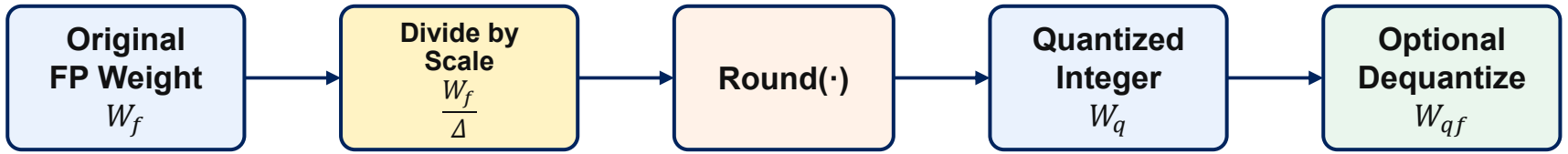
Framework	Per-Group	Flexible Precision	3-bit Accuracy	HW Efficiency	Main Limitation
FIGNA	X	Limited	Low	High	Direct mixed compute, but limited data type / precision scope
ANT	X	Limited	Low	High	Data types are mainly per-channel oriented
OliVe	X	Limited	Mid	High	Outlier protection needs victim values and special handling
Microscaling	O	O	Low	Mid	Shared exponent / FP-like path increases energy cost
BitMoD	**O**	**O**	**High**	**High**	**Target: satisfy all four goals together**

Design target **Per-Group Quantization** + **Flexible Bit-width** + **Sub-4-bit Accuracy** + **Efficient Hardware** = BitMoD motivation

Quantization Basics

Quantization Basics: Scale + Round

Integer quantization converts continuous FP weights into a small set of low-bit integer levels.



$$W_q = \text{Round}\left(\frac{W_f}{\Delta}\right)$$
$$W_{qf} = W_q \cdot \Delta$$

Core trade-off: fewer representable values → smaller memory, but larger quantization error.

Two common integer modes differ in how the real-value range is mapped to integer levels.

Symmetric quantization

0-centered grid

Asymmetric quantization

scale + zero-point

Question for this section

When is each mapping good, and what error does it create?

Symmetric Integer Quantization

Assumption: the minimum and maximum values have the same absolute range around 0.

Core equations

$$\Delta = \frac{W_{f,\max}}{2^{b-1} - 1}$$

$$W_q = \text{Round}\left(\frac{W_f}{\Delta}\right)$$

$$W_{qf} = W_q \cdot \Delta$$

Real zero maps to integer zero, so no zero-point metadata is needed.

Interpretation

1. Choose the largest absolute weight value.
2. Divide it by the largest signed integer value.
3. Use that scale Δ to round every floating-point weight.
4. Optionally multiply back by Δ for dequantization.

4-bit signed range

$$\{-7, -6, \dots, 0, \dots, +6, +7\}$$

Limitation: if the weight distribution is skewed, many integer levels may be wasted.

Symmetric Quantization Example

Example: 4-bit signed integer, |W|max = 3.5, and one original weight Wf = 1.2.

Step 1. Scaling factor

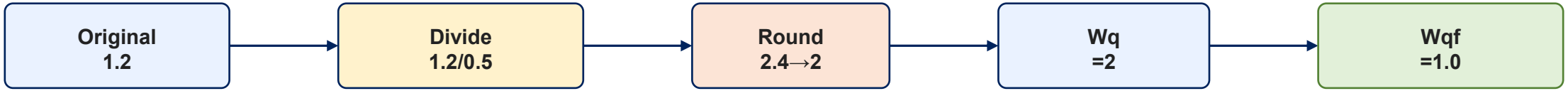
$$\Delta = \frac{3.5}{2^{4-1}-1} = \frac{3.5}{7} = 0.5$$

Step 2. Quantize one weight

$$W_q = \text{Round}\left(\frac{1.2}{0.5}\right) = \text{Round}(2.4) = 2$$

Step 3. Optional dequantization

$$W_{qf} = 2 \times 0.5 = 1.0$$



Result: 1.2 is approximated as 1.0, so the quantization error is 0.2.

Asymmetric Integer Quantization

Asymmetric quantization does not force the range to be centered at zero. It adds a zero-point z.

Core equations

$$\Delta = \frac{\text{Range}(W_f)}{2^b - 1}$$

$$z = \text{Round}\left(\frac{-W_{f,\min}}{\Delta}\right)$$

$$W_q = \text{Round}\left(\frac{W_f}{\Delta}\right) + z$$

$$W_{qf} = (W_q - z) \cdot \Delta$$

What does zero-point do?

z is the integer location that represents real 0.

This allows the integer grid to shift with the actual tensor range instead of forcing symmetry.

Example mapping idea

Better fit for skewed / asymmetric weight groups.

Asymmetric Quantization Example

Example: real range [-1.0, 3.0], 4-bit unsigned integers [0, 15], and Wf = 1.3.

Step 1. Compute scale

$$\Delta = \frac{3.0 - (-1.0)}{2^4 - 1} = \frac{4.0}{15} \approx 0.2667$$

Step 2. Compute zero-point

$$z = \text{Round}\left(\frac{1.0}{0.2667}\right) = 4$$

Step 3. Quantize one weight

$$W_q = \text{Round}\left(\frac{1.3}{0.2667}\right) + 4$$

$$= \text{Round}(4.875) + 4 = 5 + 4 = 9$$

Meaning

The +z term shifts the rounded value so that the integer grid aligns with the real range.

Therefore Wf = 1.3 becomes integer 9.

Zero-point mapping

Symmetric vs Asymmetric: Summary

The key difference is whether the integer grid is fixed around zero or shifted using a zero-point.

Aspect	Symmetric	Asymmetric
Integer range	Signed, centered at 0	Usually unsigned with zero-point
Metadata	Only scale Δ	Scale Δ + zero-point z
Best case	Balanced distribution	Skewed / shifted distribution
Trade-off	Simple, low metadata	Better fit, more metadata

Symmetric

$$W_q = \text{Round}\left(\frac{W_f}{\Delta}\right)$$

Asymmetric

$$W_q = \text{Round}\left(\frac{W_f}{\Delta}\right) + z$$

Connection to BitMoD: LLM weight groups are often locally asymmetric, so BitMoD later keeps the benefit of asymmetry using FP3/FP4 special values.

Quantization Granularity: What Shares One Scale?

Granularity decides how many weights share the same scaling factor Δ.

Weight tensor view

$$W_f \in \mathbb{R}^{K \times D}$$

K = number of output channels
D = number of weights in each channel

Three options

Granularity	What shares Δ?	Scope of outlier
Per-Tensor	Whole tensor	Whole tensor
Per-Channel	One channel	Same channel
Per-Group	Small group	Only that group

Smaller sharing scope → less outlier impact
But: more scale metadata to store

Per-Tensor Quantization: One Outlier Affects Everything

Per-Tensor uses one scaling factor for the entire weight tensor.

MD example weight tensor

Ch. 1	0.2	-0.3	0.4	-0.1	0.1	-0.2	0.3	**4.0**
Ch. 2	0.5	-0.4	0.2	-0.1	0.6	-0.5	0.4	-0.3

4.0 is the absolute maximum, so the whole tensor must use a large Δ.

$$\Delta = \frac{4.0}{2^{4-1}-1} = \frac{4.0}{7} \approx 0.5714$$

Effect on a small weight

Take a normal small weight from the tensor:

$$W_f = 0.3$$

$$W_q = \text{Round}\left(\frac{0.3}{0.5714}\right) = 1$$

$$W_{qf} = 1 \times 0.5714 = 0.5714$$

0.3 is approximated as 0.5714
Error ≈ 0.2714

Per-Channel Quantization: Better, But Still Coarse

Per-Channel assigns a separate scaling factor to each output channel.

Each channel has its own Δ

Ch. 1	0.2	-0.3	0.4	-0.1	0.1	-0.2	0.3	**4.0** → Δ_1
Ch. 2	0.5	-0.4	0.2	-0.1	0.6	-0.5	0.4	-0.3 → Δ_2

Channel	Abs. max	Scale
Ch. 1	4.0	0.5714
Ch. 2	0.6	0.0857

Remaining problem

Per-Channel prevents the outlier from affecting other channels.

But within Channel 1, every weight still uses:

$$\Delta_1 = \frac{4.0}{7} \approx 0.5714$$

Therefore 0.2, −0.3, 0.4, −0.1 in the same channel are still affected by the 4.0 outlier.

Channel-level is better than tensor-level, but too coarse for large LLM hidden dimensions.

Per-Group Quantization: Limit the Outlier Impact

Per-Group splits each channel into smaller groups, and each group gets its own scaling factor.

General rule

$$\text{Groups/channel} = \frac{D}{G}$$

G = group size.
Recent LLM quantization frameworks often use G = 128 to balance accuracy and metadata overhead.

Group 1 Δ_1	Group 2 Δ_2

Outlier is contained in the group where it exists.

MD example: Channel 1, G = 4

G1	0.2	-0.3	0.4	-0.1	→ $\Delta_1 = 0.4/7 \approx 0.0571$
G2	0.1	-0.2	0.3	**4.0**	→ $\Delta_2 = 4.0/7 \approx 0.5714$

Group 1 can use a much smaller scale, because the 4.0 outlier is only in Group 2.

Key idea: Per-Group improves accuracy by reducing the maximum value/range inside each quantized block.

Same Weight, Different Granularity

Compare the same weight Wf = 0.3 when it belongs to Group 1.

Per-Tensor

$$\Delta = 0.5714$$

$$W_q = \text{Round}\left(\frac{0.3}{0.5714}\right) = 1$$

$$W_{qf} = 0.5714$$

Error ≈ 0.2714

Per-Channel

$$\Delta = 0.5714$$

Channel 1 still contains the 4.0 outlier.

$$W_{qf} = 0.5714$$

Error ≈ 0.2714

Per-Group

$$\Delta = 0.0571$$

$$W_q = \text{Round}\left(\frac{0.3}{0.0571}\right) = 5$$

$$W_{qf} = 5 \times 0.0571 \approx 0.2855$$

Error ≈ 0.0145

Method	Scale Δ	Approx. value	Error
Per-Tensor	0.5714	0.5714	0.2714
Per-Channel	0.5714	0.5714	0.2714
Per-Group	**0.0571**	**0.2855**	**0.0145**

Why Smaller Scaling Factor Helps

The quantization error is proportional to the scaling factor Δ. Smaller Δ means denser representable values.

Error relation

$$\text{Error}(W_{qf}) = \text{Error}_{\text{Round}}\left(\frac{W_f}{\Delta}\right) \cdot \Delta$$

$$\text{Quantization Error} \propto \Delta$$

Rounding error is unavoidable, but multiplying it by a large Δ makes the final dequantized error larger.

Therefore, reducing the local max/range is directly useful.

Scale spacing intuition

Large Δ = sparse grid

Small Δ = dense grid

Trade-off: Per-Group needs more scale metadata, but group size 128 is used to balance accuracy and overhead.

Motivation Takeaway: Why BitMoD Focuses on Per-Group

Granularity determines the accuracy–metadata trade-off before introducing BitMoD's new data types.

Per-Tensor

One Δ for whole tensor

Outlier affects every weight

Low metadata, low accuracy

Per-Channel

One Δ per output channel

Outlier affects the same channel

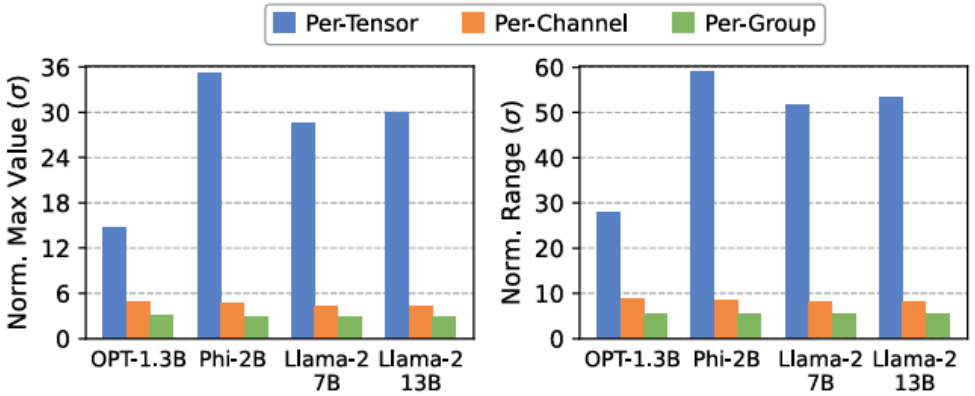
Middle ground

Per-Group

One Δ per small group

Outlier is localized

Higher metadata, best accuracy



BitMoD direction: use Per-Group Quantization for accuracy, then design hardware to make per-group dequantization efficient.

Quantization Data Type Matters: Same 4-bit, Different Error

Even with the same 4-bit precision, the data type changes quantization levels, error pattern, and perplexity.

INT4-Sym

Zero-centered integer range
Simple, but rigid

… -2 -1 0 +1 +2 …

INT4-Asym

Integer + zero-point
Better for shifted range

zero-point z shifts range

FP4

Floating-point levels
Dense near zero

0, ±0.5, ±1, ±2, ±4…

Flint

Power-of-two-like levels
Hardware-friendly

levels chosen for cheap compute

TABLE I. Wikitext-2 perplexity (↓) under different quantization granularity and 4-bit data types. "PC" and "PG" stand for per-channel and per-group, respectively. The group size is 128.

Model	OPT-1.3B		Phi-2B		Llama-2-7B		Llama-2-13B	
Granularity	PC	PG	PC	PG	PC	PG	PC	PG
FP16	14.62	14.62	9.71	9.71	5.47	5.47	4.88	4.88
INT4-Sym	36.05	16.04	13.03	11.15	12.92	5.84	5.47	5.07
INT4-Asym	48.41	15.41	12.08	**10.67**	8.89	**5.77**	5.27	**5.01**
FP4	16.07	**14.99**	**11.24**	10.68	8.07	**5.77**	**5.15**	5.05
Flint	**15.87**	16.23	11.71	11.23	**6.67**	6.09	5.31	5.29

Result 1

Flint is strong in some PC cases, but not consistently best in PG.

Result 2

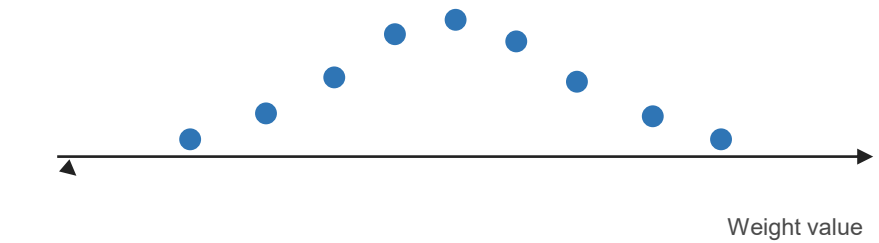
INT4-Asym and FP4 are strong in PG → FP + asymmetry are both useful.

Why FP and Asymmetry Help Per-Group Quantization

BitMoD combines two observations: LLM weights are Gaussian-like, and small groups can be asymmetric.

1. Floating Point

Gaussian-like weight distribution

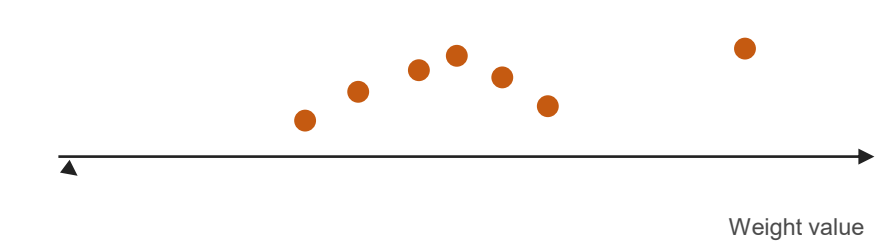


Many weights are concentrated near zero.
FP levels can be denser around small magnitudes than uniform integers.

Small values need finer resolution

2. Asymmetry

One-sided group outlier



A small weight group can have a positive or negative outlier on only one side.
Symmetric levels waste range on the empty side.

Use range where the group needs it

BitMoD Insight: Reuse Redundant Zero as a Special Value

Low-precision floating point uses sign-magnitude, so +0 and −0 are two bit patterns for the same numerical zero.

Basic FP3 unique values

$$\{0, \pm 1, \pm 2, \pm 4\}$$

3-bit gives 8 bit patterns, but +0 and −0 both represent numerical zero.

$$\frac{1}{8} = 12.5\%$$

In FP3, one out of eight patterns is redundant zero.
This is too expensive to waste.

BitMoD replacement

Basic FP3
0, ±1, ±2, ±4

↓

Replace redundant −0
with one Special Value

↓

Example
0, ±1, ±2, ±4, +6

Special Value can improve resolution or extend one-sided range for each weight group.

Quantization Bit-width Matters: 8/4-bit Is Not Enough

Different bit-widths provide different accuracy–efficiency trade-offs. 6-bit is important, and 3-bit needs better data types.

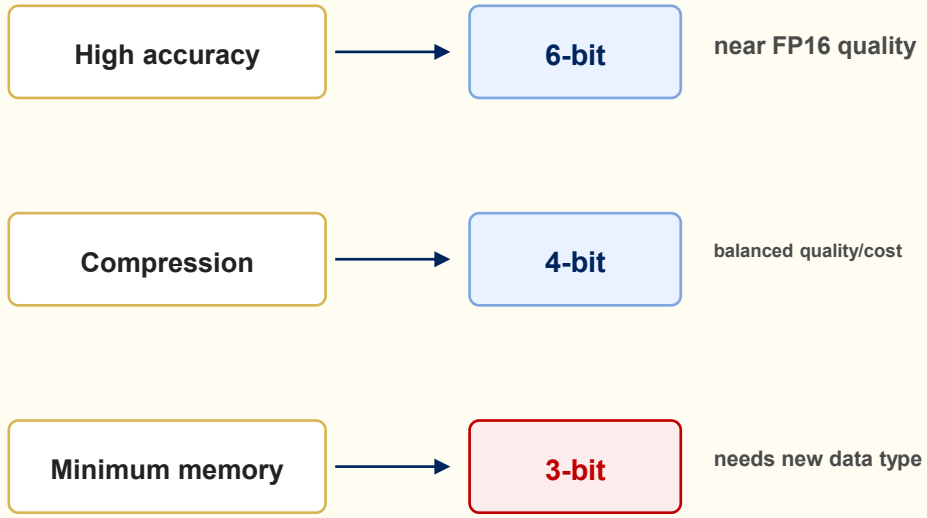
6-bit can be almost lossless

TABLE II. Wikitext-2 and C4 perplexity (↓) under different 6-bit data types. We use per-group weight quantization with a group size of 128.

Model	OPT-1.3B		Phi-2B		Llama-2-7B		Llama-2-13B	
Dataset	Wiki	C4	Wiki	C4	Wiki	C4	Wiki	C4
FP16	14.62	14.72	9.71	12.74	5.47	6.97	4.88	6.47
INT6-Sym	**14.51**	14.80	9.85	12.82	**5.49**	**6.99**	**4.89**	**6.46**
INT6-Asym	14.61	14.78	**9.76**	**12.8**	**5.49**	**6.99**	**4.89**	**6.46**
FP6-E2M3	14.59	**14.76**	9.85	**12.8**	5.52	**6.99**	4.92	6.49
FP6-E3M2	14.81	14.81	9.81	12.87	**5.49**	7.02	**4.89**	6.50

INT6-Sym average PPL loss is reported as < 0.05.
So 6-bit is a useful middle point between 8-bit and 4-bit.

Precision choice should be flexible



Fixed 8/4-bit hardware cannot fully exploit this trade-off.

Bit-Serial Architecture: Flexible Precision by Term Count

A bit-serial design can use one PE structure while changing the number of processed terms by precision.

Bit-Parallel limitation

Hardware datapath is often specialized for fixed precision.

INT8 | INT4 | ?

6-bit and 3-bit are awkward to support efficiently.

Good for common fixed widths, but less flexible.

Bit-Serial idea

Data Type	Bit-serial Terms
INT8	4
INT6	3
FP4	2
FP3	2

Lower precision → fewer terms → fewer cycles.
The same PE can support multiple bit-widths by changing term count.

This motivates BitMoD's unified bit-serial representation.

But Existing Bit-Serial Accelerators Are Not Enough

Bit-serial flexibility alone does not solve LLM quantization. Two missing pieces remain.

Problem 1. Mostly integer data types

Existing bit-serial accelerators are mainly designed around integer representation.

↓

At 3-bit, simple integer levels can cause large accuracy loss.

Need: FP/asymmetric data type suitable for local group distribution.

Problem 2. Per-Group dequantization cost

Each group has a different scaling factor:

$\Delta_1, \Delta_2, \Delta_3, \ldots$

So every group partial sum must be dequantized separately.

↓

FP scale requires FP unit → area/energy overhead.

Motivation Summary: What BitMoD Must Satisfy

Existing frameworks satisfy only part of the requirement. BitMoD is motivated by satisfying all four together.

TABLE III. Comparison between *BitMoD* and SOTA co-design frameworks for LLM acceleration

Framework	Per-group Quant?	Supported Precision	Accuracy @ 3-bit Weight	Hardware Efficiency
AWQ [30]	Yes	Limited	High	Low
FIGNA [27]	No	Limited	Low	High
ANT [26]	No	Limited	Low	High
OliVe [25]	No	Limited	Medium	High
Microscaling [40]	Yes	Many	Low	Medium
***BitMoD* (Ours)**	Yes	Many	High	High

Core motivation

Design the quantization format and the accelerator together, so low-bit weights reduce both memory traffic and actual compute cost.

BitMoD Quantization Framework

A. Asymmetric FP3

Because sign-magnitude FP stores +0 and −0 as different bit patterns, one quantization level is duplicated.

3-bit sign-magnitude pattern view

Example conceptual FP3 encoding

S	E/M	Stored	Value	
0	00	+0	0	two patterns same value 0
1	00	−0	0	
0	01	+1	+1	
1	01	−1	−1	
0	10	+2	+2	
1	10	−2	−2	
0	11	+4	+4	
1	11	−4	−4	

Actual unique FP3 values

$$\{0, \pm 1, \pm 2, \pm 4\}$$

Total bit patterns $2^3 = 8$

Unique numeric levels **7 levels**

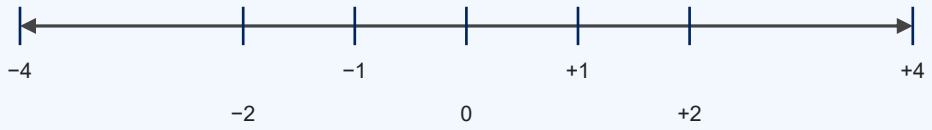
BitMoD turns the wasted −0 pattern into one useful Special Value.

From Redundant −0 to Special Value

BitMoD uses the redundant negative zero pattern as an extra quantization level selected for each weight group.

Before: Basic FP3

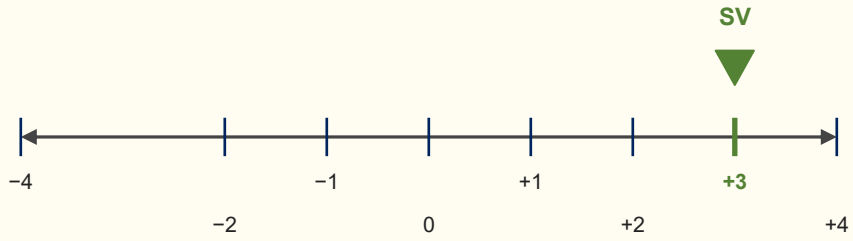
+0 and −0 both map to 0



Only 7 unique values are available, even though there are 8 bit patterns.

After: Extended FP3

−0 encoding is reinterpreted as a Special Value



The group now has 8 useful quantization levels.

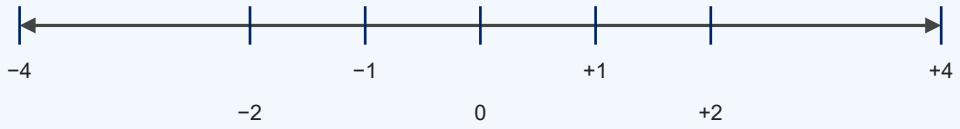
Important: the Special Value is not chosen in this figure; it is defined here as a candidate and later selected per group by Algorithm 1.

FP3-ER: Extended Resolution

ER adds a Special Value inside the original FP3 range, making the value grid denser.

Basic FP3

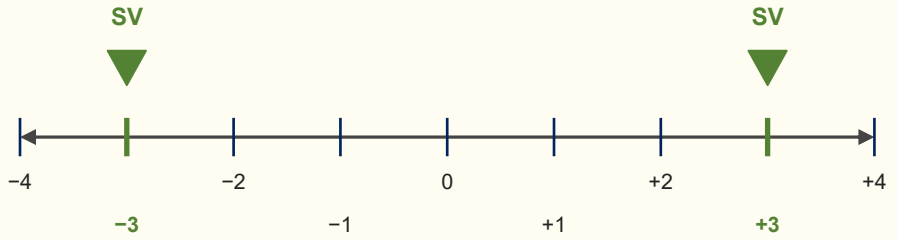
Range stays: $[-4, +4]$



Notice the empty space between +2 and +4.

FP3-ER

Special Value: $\mathrm{SV} = \pm 3$



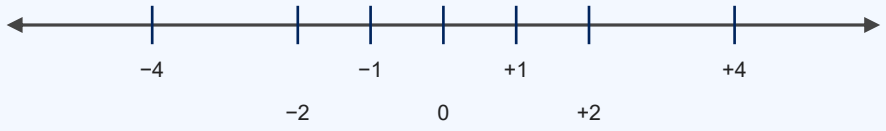
Purpose: improve quantization resolution without changing the maximum range.

Useful for symmetric, Gaussian-like weight groups.

FP3-EA: Extended Asymmetry

EA adds a Special Value outside the original FP3 range, allowing one side to represent a larger value.

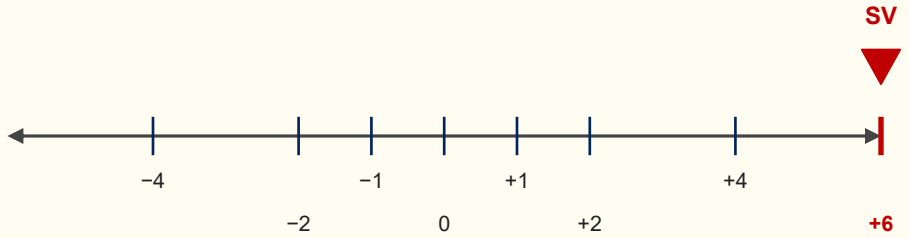
Basic FP3



Negative Range → −4 **Positive Range → +4**

FP3-EA (+6 example)

Special Value: $\mathrm{SV} = \pm 6$



Negative Range → −4 **Positive Range → +6**

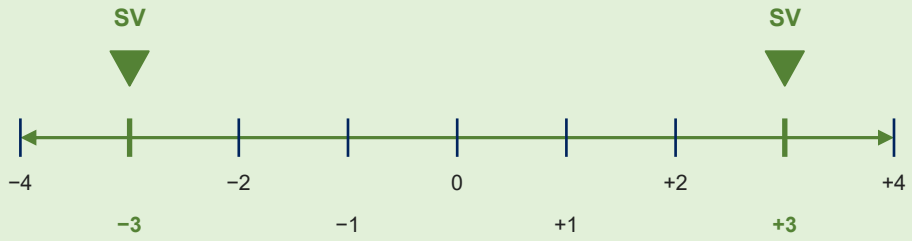
Purpose: represent one-sided outliers or shifted weight groups.

ER vs EA: Different Special Values for Different Weight Groups

BitMoD prepares both resolution-oriented and asymmetry-oriented extensions.

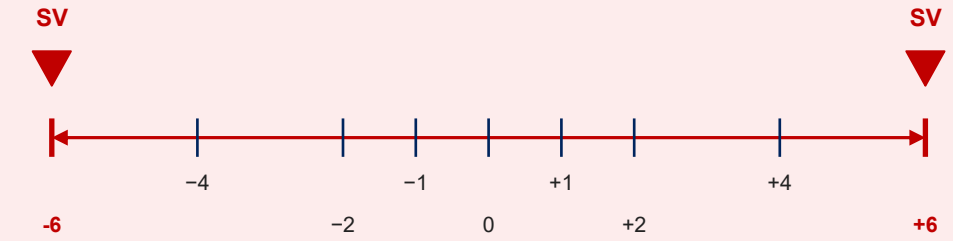
Type	Special Value	Range Change	Best for
FP3-ER	±3	Range X	Dense / Gaussian-like group
FP3-EA	±6	Extend Range	Asymmetric outlier group

ER intuition



Adds missing levels inside the same range.

EA intuition



Extends the range toward the outlier side.

The same basic FP3 can become ER or EA depending on the selected Special Value.

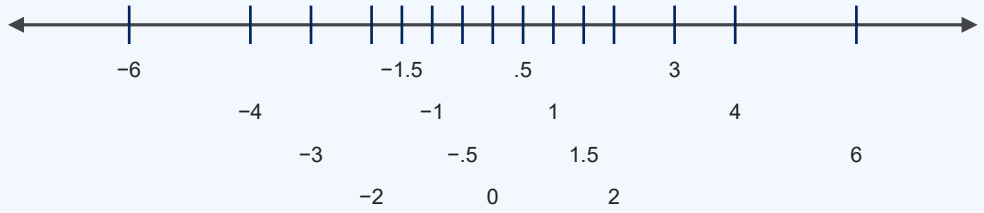
FP4 Extension: Same Idea, More Levels

BitMoD applies the same redundant-zero reuse idea to FP4.

Basic FP4 values

$$\{0, \pm 0.5, \pm 1, \pm 1.5, \pm 2, \pm 3, \pm 4, \pm 6\}$$

Basic FP4 already has more representable values than FP3, including fractional levels such as ±0.5 and ±1.5.



BitMoD FP4 Special Values

Extended Dtype	Special Value	Role
FP4-ER	−5 or +5	Resolution Extend
FP4-EA	−8 or +8	Asymmetry Extend

±5 fills the gap between ±4 and ±6.

±8 extends the dynamic range beyond ±6.

Final Candidate Data Types Used by BitMoD

At this point, BitMoD has defined four Special Value candidates for each precision. The next section chooses one per group.

Basic Dtype	Extended Dtype	Special Value	Purpose
FP3	FP3-ER	−3 or +3	Resolution Extend
FP3	FP3-EA	−6 or +6	Asymmetry Extend
FP4	FP4-ER	−5 or +5	Resolution Extend
FP4	FP4-EA	−8 or +8	Asymmetry Extend

Important distinction

This section defines candidate data types.
It does not yet decide which one a group uses.

→

Next: Fine-grained Adaptation

Algorithm 1 tests these candidates per group and selects the one with the lowest MSE.

Core message: BitMoD combines FP's Gaussian-like representation with asymmetric special values using a hardware-friendly encoding.

B. Fine-grained Data Type Adaptation: Motivation

Asymmetric FP3/FP4 defines candidate Special Values. Fine-grained adaptation decides which one each weight group should use.

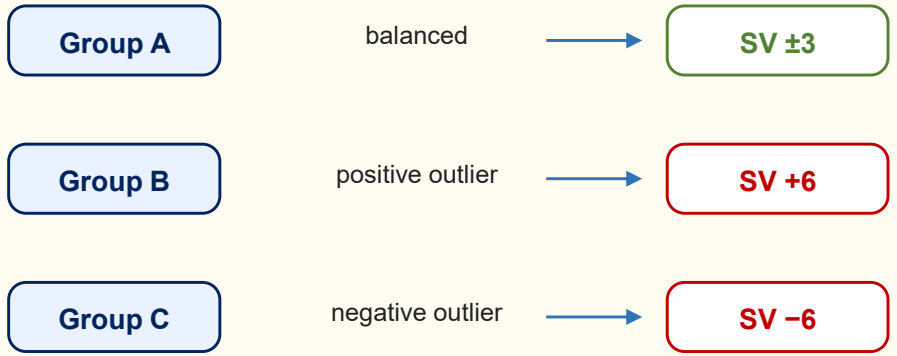
Previous section: candidate values

Precision	Basic Values	Special Values
FP3	{0, ±1, ±2, ±4}	{-3, +3, -6, +6}
FP4	{0, ±0.5, ±1, ±1.5, ±2, ±3, ±4, ±6}	{-5, +5, -8, +8}

However, one group does not use all four Special Values.
It selects only one.

Fine-grained adaptation

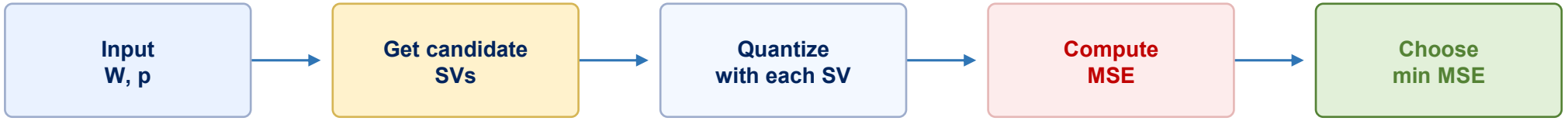
Same tensor, different local distributions



Not only the scaling factor, but also the data type is adapted per group.

Algorithm 1: Search the Best Special Value

For each weight group, BitMoD tries every candidate Special Value and chooses the one with the smallest MSE.



Input / Output

$$W, p \rightarrow \left(W_{qout}, v_{out}\right)$$

Output contains both quantized weights and the selected SV.

Core objective

$$v_{out} = \arg\min_{v \in \mathcal{V}_{SV}} \mathrm{MSE}\left(W, W_q^{(v)}\right)$$

Choose the Special Value that minimizes quantization error.

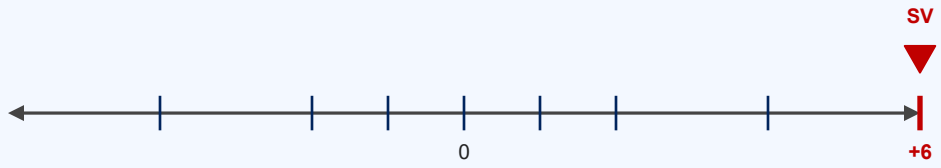
Key idea: exhaustive candidate search is simple because there are only 4 Special Values per precision.

Non-linear Quantization and Error Measurement

Because FP3/FP4 levels are not equally spaced like integers, BitMoD uses non-linear quantization and evaluates MSE.

Non-linear levels

FP3-EA(+6) example



Spacing is uneven: −4, −2, −1, 0, +1, +2, +4, +6

$$W_q = \text{NonLinearQuantize}(W, Q_v)$$

Error criterion

For each candidate, compare the original group W and the quantized group Wq.

$$\text{MSE}(W, W_q) = \frac{1}{N}\sum_{i=1}^{N}(W_i - W_{q,i})^2$$

Small MSE means the quantized levels fit the current group better.

Algorithm 1 chooses the SV with the smallest MSE.

Numerical Example: Selecting +6 for a Positive Outlier Group

This group has a large positive weight 5.7, so the +6 candidate fits best.

Weight group

$$W = [-0.8,\ 0.3,\ 1.1,\ 2.2,\ 3.8,\ 5.7]$$

MSE comparison

Special Value	Data Type Meaning	MSE
−3	Negative resolution	0.42
+3	Positive resolution	0.31
−6	Negative range expansion	0.57
+6	**Positive range expansion**	**0.08**

Decision

$$v_{out} = +6$$

Final quantization level set for this group:

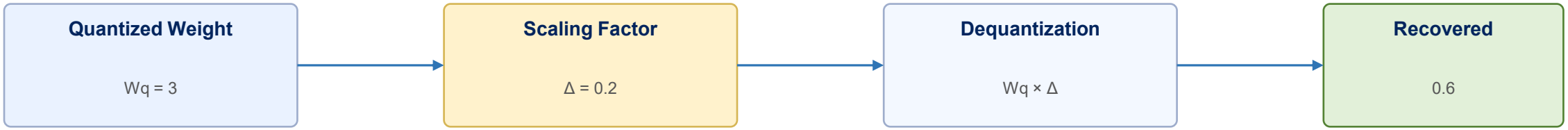
$$\{0, \pm1, \pm2, \pm4, +6\}$$

This is FP3-EA(+6), which expands the positive side to represent the outlier.

If the outlier were negative, −6 could be selected instead; if the group were balanced, ±3 may be better.

C. Quantized weights need dequantization

Quantized weight values are integer / low-bit codes, not the final real-valued weights used in computation.



General formula

$$W_{qf} = W_q \Delta$$

scale must be applied after quantization

Small numeric example

$$W_q = 3,\ \Delta = 0.2 \Rightarrow W_{qf} = 3 \times 0.2 = 0.6$$

same idea used for group partial sums

Core intuition

Quantization saves memory, but dequantization is still needed somewhere in the hardware datapath.

Per-channel dequantization is simple

In per-channel quantization, every weight in the same channel shares one scaling factor.

Example

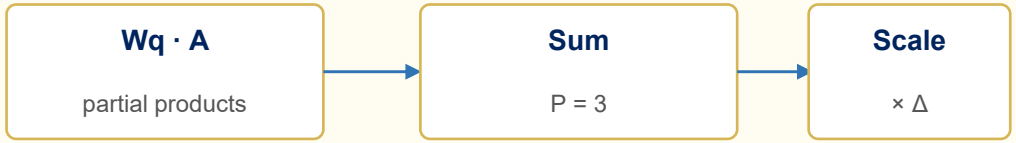
$$W_q = [2, -1, 3, 1], \quad A = [1.0, 2.0, 0.5, 1.5], \quad \Delta = 0.2$$

$$P = 2(1.0) + (-1)(2.0) + 3(0.5) + 1(1.5) = 3$$

$$Y = P\Delta = 3 \times 0.2 = 0.6$$

Compute dot-product first → apply one scale once.

Why this is hardware-friendly



Only one scaling factor exists for the entire channel, so the re-scaling can be delayed until after accumulation.

No per-group scale switching
No repeated floating-point re-scaling
Accumulator can work on one shared scale

Per-group dequantization cannot be postponed

In per-group quantization, one channel is split into groups, and each group has a different scaling factor.

Group 1

$$W_q = [2, -1], \quad \Delta_1 = 0.1$$

$$P_1 = 2(1.0) + (-1)(2.0) = 0, \quad P_1\Delta_1 = 0$$

Group 2

$$W_q = [3,1], \quad \Delta_2 = 0.4$$

$$P_2 = 3(0.5) + 1(1.5) = 3, \quad P_2\Delta_2 = 1.2$$

Final output

$$Y = P_1\Delta_1 + P_2\Delta_2 = 0 + 1.2 = 1.2$$

Different groups use different scales, so P1 and P2 must be scaled separately.

Cannot use one final scale after merging all groups

Scaling factor quantization

Scaling factor quantization: detailed example

Step 1. Original group scales

$$[\Delta_1, \Delta_2, \Delta_3, \Delta_4] = [0.10, 0.21, 0.32, 0.40]$$

The largest scale is 0.40.

Step 2. Build ΔSF

$$\Delta_{SF} = \frac{0.40}{127} \approx 0.00315$$

127 is the positive maximum of signed INT8 symmetric quantization.

Step 3. Convert to INT8

$$SF_{q,1} = \text{Round}\left(\frac{0.10}{0.00315}\right) \approx 32$$

$$[SF_{q,1}, SF_{q,2}, SF_{q,3}, SF_{q,4}] = [32, 67, 102, 127]$$

$$\Delta_g \approx SF_{q,g}\Delta_{SF}$$

Example: 32 × 0.00315 ≈ 0.1008 ≈ 0.10

Rewriting dequantization for hardware

Once Δg is approximated as SFq,g × ΔSF, the common factor can be moved outside the group summation.

Mathematical transformation

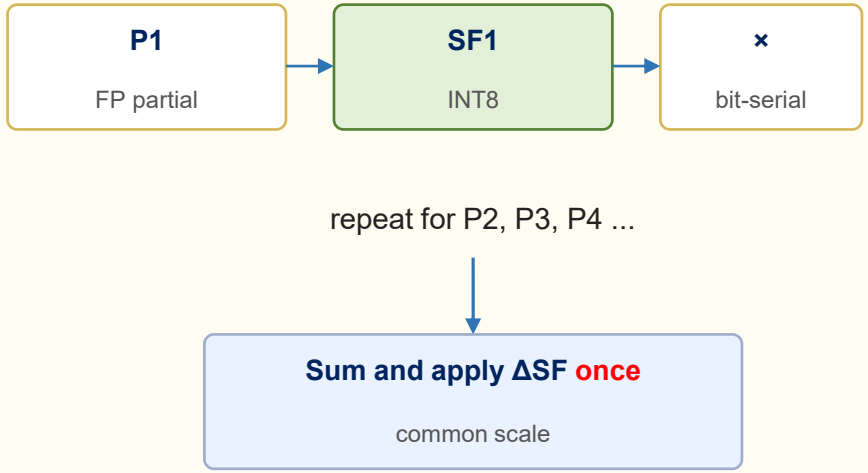
$$Y = P_1\Delta_1 + P_2\Delta_2 + P_3\Delta_3 + P_4\Delta_4$$

$$\Delta_g \approx SF_{q,g}\Delta_{SF}$$

$$Y \approx \Delta_{SF}\left(P_1SF_{q,1} + P_2SF_{q,2} + P_3SF_{q,3} + P_4SF_{q,4}\right)$$

Group-specific multiplication now uses INT8 SFq,g, not FP16 Δg.

Hardware view



This is why scaling factor quantization is connected to the BitMoD hardware design.

Why INT8 scales are bit-serial friendly

An INT8 scale can be consumed one bit per cycle and implemented with shift-and-add rather than a full FP multiplier.

Example: INT8 scale = 13

$$13 = 00001101_2 = 8 + 4 + 1$$

Active bits are 0, 2, and 3.

Shift-and-add form

$$m_{ACC} \times 13 = m_{ACC} + (m_{ACC} \ll 2) + (m_{ACC} \ll 3)$$

Cycle-wise interpretation

bit0 = 1 → use mACC
bit1 = 0 → skip
bit2 = 1 → use mACC << 2
bit3 = 1 → use mACC << 3

The example scale 13 is illustrative; the paper uses INT8 per-group scaling factors generally.

Why BitMoD chooses INT8 scaling factors

The paper sweeps scaling-factor precision under INT4-asymmetric per-group quantization with group size 128.

TABLE V. Wikitext-2 and C4 perplexity (↓) under different precision for the per-group scaling factor (SF). We use `INT4-Asym` for weight quantization with a group size of 128.

Model	OPT-1.3B		Phi-2B		Llama-2-7B		Llama-2-13B	
SF Bits	Wiki	C4	Wiki	C4	Wiki	C4	Wiki	C4
FP16	15.41	15.84	10.68	13.66	5.77	7.31	5.01	6.62
INT8	15.41	15.84	10.68	13.66	5.77	7.31	5.01	6.62
INT6	15.43	15.84	10.74	13.71	5.77	7.31	5.01	6.62
INT4	15.52	15.93	10.76	13.73	5.77	7.36	5.03	6.64
INT2	18.46	18.61	15.68	18.32	8.41	10.63	6.19	8.12

Observation 1

INT8 matches FP16 exactly in the table.

Observation 2

INT6 / INT4 are close, but not identical.

Observation 3

INT2 hurts perplexity significantly.

Channel size, group size, and metadata overhead

BitMoD stores one INT8 scaling factor and one 2-bit Special Value ID per group.

How many groups per channel?

$$\frac{D}{G} = \frac{4096}{128} = 32 \text{ groups/channel}$$

Example from the MD file

Channel size D = 4096
Group size G = 128

One channel has 32 groups.
Each group has its own INT8 scaling factor.

Metadata per group

$$8 \text{ bits} + 2 \text{ bits} = 10 \text{ bits/group}$$

8-bit INT8 scale
+
2-bit SV ID
=
10 bits / group

Overhead vs weight bits

$$\frac{10}{128 \times 3} \times 100 \approx 2.6\%$$

$$\frac{10}{128 \times 4} \times 100 \approx 1.95\%$$

With G = 128, metadata overhead is small compared with the weight storage.

BitMoD Harware Accelerator

A. Unified Bit-serial Representation

BitMoD hardware should support multiple weight formats in a single accelerator without losing low-bit compute efficiency.

INT8
high accuracy

INT6
accuracy-cost tradeoff

FP4
compression

FP3
high efficiency

Problem 1: different representation

INT8 / INT6
→ Integer Representation

FP4 / FP3
→ Floating-Point Representation

Problem 2: naïve solutions fail

Separate operators

→ hardware area and complexity increase

Convert all to INT8

→ FP3/FP4 lose low-bit compute benefit

BitMoD answer

INT / FP → common term

Sign | Exp | Man | Bsig

Same PE, fewer terms for lower precision

Unified Bit-serial Term: the common language for all data types

Every supported weight is decomposed into one or more terms with the same four fields.

Field	Meaning	Role inside PE
Sign	value sign	positive / negative contribution
Exp	exponent	power-of-two scale inside the term
Man	mantissa	whether the term is active
Bsig	bit-significance	original bit position / shift amount

Term value

$$V_{term} = (-1)^{sign} \cdot 2^{exp} \cdot man \cdot 2^{bsig}$$

$$W = T1 + T2 + \cdots + Tn$$

Core intuition

Lower bit-width → fewer bit-serial terms → fewer PE cycles

So quantization reduces not only memory footprint, but also the amount of computation.

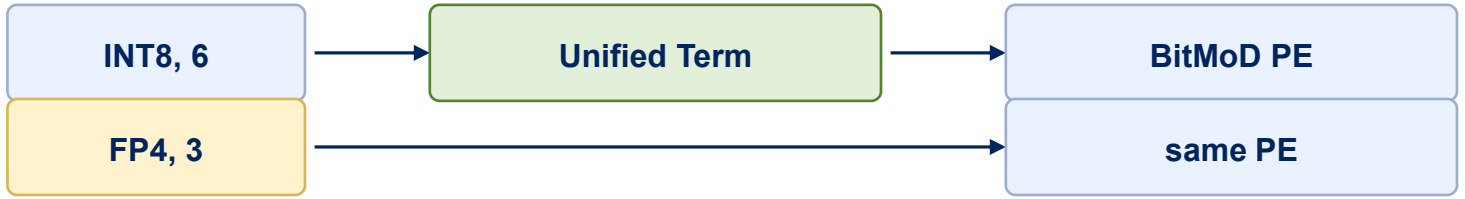
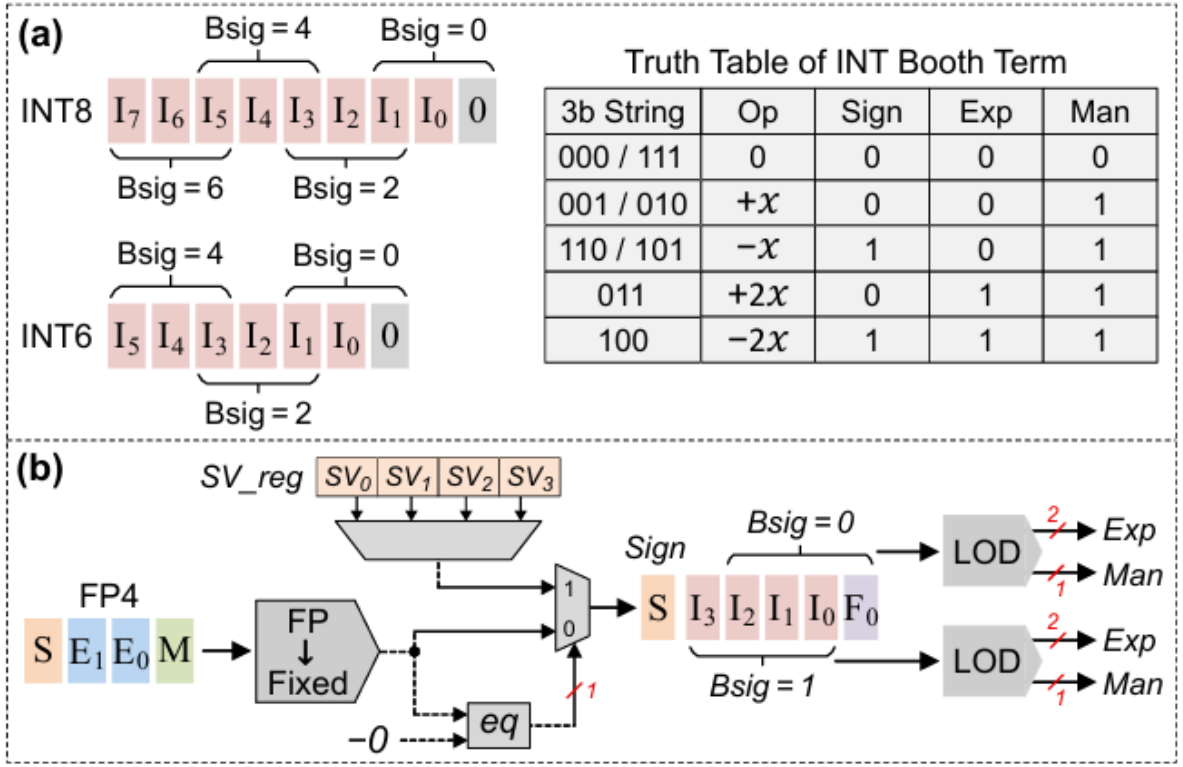


Figure 4 Overview: two decoders, one output format

INT formats and FP formats enter different front-end decoders, but both produce Sign / Exp / Man / Bsig terms.



Truth Table of INT Booth Term

3b String	Op	Sign	Exp	Man
000 / 111	0	0	0	0
001 / 010	$+x$	0	0	1
110 / 101	$-x$	1	0	1
011	$+2x$	0	1	1
100	$-2x$	1	1	1

Fig. 4: Unified bit-serial representation of (a) INT8, INT6, and (b) FP4. Every bit-serial term contains four parts: sign, exponent (exp), mantissa (man) and bit-significance (bsig).

Figure 4(a): INT8 / INT6

Booth Encoding decomposes integer binary strings into 3-bit Booth strings.

Figure 4(b): FP4 / FP3

FP → Fixed conversion + redundant −0 handling + LOD creates at most 2 terms.

Final output of both paths

Unified Bit-serial Terms: Sign | Exp | Man | Bsig

INT8	INT6	FP4	FP3
4 terms	3 terms	≤2 terms	≤2 terms

Case 1 — INT8: Booth Encoding creates 4 terms

INT8 has 8 binary bits, so radix-4 Booth encoding produces four 3-bit Booth strings with Bsig = 0, 2, 4, 6.

Example: INT8 weight 45

$45 \rightarrow 00101101_2$

0	0	1	0	1	1	0	1
I7	I6	I5	I4	I3	I2	I1	I0

Booth terms generated from LSB side

Booth string	Op	Bsig	Contribution
010	+x	0	+1
110	−x	2	−4
101	−x	4	−16
001	+x	6	+64

Result

$$45 = +64 - 16 - 4 + 1$$

Each contribution becomes one unified bit-serial term.

Term	Sign	Exp	Man	Bsig
$+x \cdot 2^6$	0	0	1	6
$-x \cdot 2^4$	1	0	1	4
$-x \cdot 2^2$	1	0	1	2
$+x \cdot 2^0$	0	0	1	0

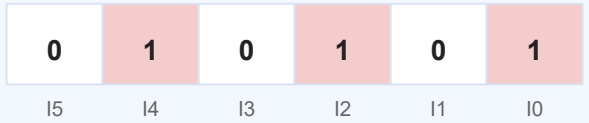
INT8 → 4 cycles for 4 bit-serial terms

Case 2 — INT6: same Booth Encoding, fewer terms

INT6 uses the same Booth decoder as INT8, but the shorter bit-width reduces the number of terms from four to three.

Example: INT6 weight 21

$21 \rightarrow 010101_2$



Booth terms

Booth string	Op	Bsig	Contribution
010	+x	0	+1
010	+x	2	+4
010	+x	4	+16

Result

$$21 = +16 +4 +1$$

Only three terms are required because INT6 has three radix-4 groups.

Data Type	Bsig values	Terms
INT8	6, 4, 2, 0	4
INT6	4, 2, 0	3

Takeaway: lower integer precision directly reduces bit-serial cycles.

Booth Truth Table: how INT bits become Sign / Exp / Man

A 3-bit Booth string is decoded into one of five simple operations: 0, +x, −x, +2x, −2x.

3-bit String	Operation	Sign	Exp	Man
000 / 111	0	0	0	0
001 / 010	+x	0	0	1
110 / 101	−x	1	0	1
011	+2x	0	1	1
100	−2x	1	1	1

Example 1: 011

$$011 \rightarrow +2x$$

Sign = 0, Exp = 1, Man = 1

Example 2: 100

$$100 \rightarrow -2x$$

Sign = 1, Exp = 1, Man = 1

Bsig is the second scale inside the term

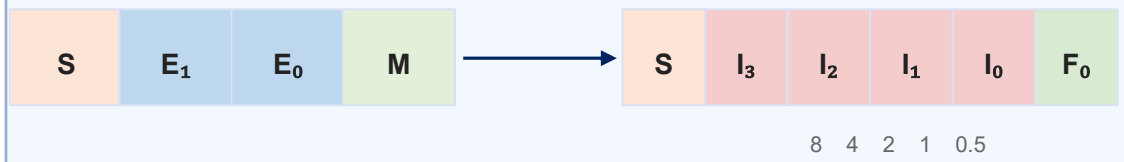
$$V_{term} = (-1)^{Sign} \times 2^{Exp} \times Man \times 2^{Bsig}$$

Same +x has different contribution when Bsig changes: +x·2^0, +x·2^2, +x·2^4, ...

Case 3 — FP4: FP → Fixed, then LOD

FP4 cannot use the integer Booth path directly, so BitMoD first converts FP4 to a sign-magnitude fixed-point value.

FP4 input format



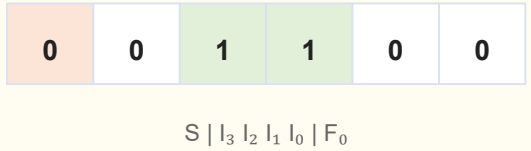
Why these bits?

I_3 is needed for FP4-EA special value ±8.

F_0 is needed for values such as ±0.5 and ±1.5.

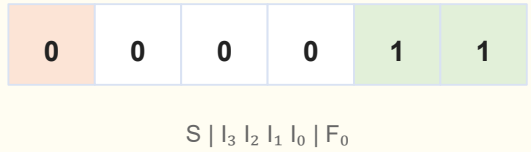
Example: FP4 value 6

$$6 = 4 + 2$$



Example: FP4 value 1.5

$$1.5 = 1 + 0.5$$



Case 4 — FP4 Special Value: −0 triggers SV_reg

The redundant negative-zero encoding is not treated as zero; it becomes a group-specific Special Value selected by metadata.

Example mapping in FP4

2-bit ID	Selected SV
00	−5
01	+5
10	−8
11	+8

Important distinction

Algorithm 1 already selected the best SV offline.

Figure 4 only uses the stored metadata to choose one value from SV_reg.

No MSE search happens inside this decoder.

Case 5 — LOD converts FP4 fixed value into at most two terms

Extended FP4 values have at most two 1-bits after fixed-point conversion, so two LOD blocks are enough.

Example: +6

0 | 0 1 1 0 | 0

LOD finds I_2 and I_1 → Term 1 = 4, Term 2 = 2

Example: +1.5

0 | 0 0 0 1 | 1

LOD finds I_0 and F_0 → Term 1 = 1, Term 2 = 0.5

Why two LOD blocks?

LOD 1 checks $\{I_3, I_2, I_1, I_0\}$

LOD 2 checks $\{I_2, I_1, I_0, F_0\}$

Each detected 1-bit becomes a Sign / Exp / Man term sent to the PE.

Case 6 — FP3: reuse the FP4 decoder

Extended FP3 values are a subset of the values representable by the FP4 fixed-point decoder.

Basic FP3

$$\{0, \pm 1, \pm 2, \pm 4\}$$

Example: +4 → fixed bits 0 | 0 1 0 0 | 0 → one active term

FP3 Special Values

$$\text{FP3-ER: } \pm 3 \quad \text{FP3-EA: } \pm 6$$

Example: +6 → same FP4 fixed path → 4 + 2

Hardware implication

No separate FP3 decoder is needed.

FP3 and FP4 share the same FP → Fixed + LOD hardware and both produce at most 2 terms.

Programmable Special Value: fixed hardware, flexible candidate values

SV_reg does not have to permanently store only one hard-coded set of values; it can be programmed for the target model.

Default candidates in the paper

Precision	SV candidates
FP3	−3, +3, −6, +6
FP4	−5, +5, −8, +8

These four choices are selected so only 2-bit metadata is needed per group.

Example: if SV = 7 is desired

Naïve decomposition

$$7 = 4 + 2 + 1 \rightarrow 3 \text{ terms}$$

Optimized decomposition

$$7 = 8 - 1 \rightarrow 2 \text{ terms}$$

The decoder can be slightly modified to keep the number of bit-serial terms small.

Unified Representation Summary

The purpose of Figure 4 is to make every supported data type look the same to the BitMoD PE.

Case	Input representation	Conversion method	Terms
INT8	8-bit integer	Booth Encoding	4
INT6	6-bit integer	Booth Encoding	3
FP4	S | E_1E_0 | M	FP→Fixed + −0/SV + LOD	≤2
FP3	subset of FP4	reuse FP4 decoder	≤2

Why this matters

One PE can support all target data types.

Performance link

Fewer terms mean fewer bit-serial cycles.

Next section

BitMoD PE computes these terms with FP16 activations.

Core message: low-bit data types become hardware-efficient only after they are converted into a common bit-serial form.

B. BitMoD Processing Element

BitMoD PE directly computes mixed-precision dot products between bit-serial low-precision weights and FP16 activations.

PE input format

Weight term from Figure 4

$$w = (w_s, w_e, w_m, w_{bsig})$$

FP16 activation already has FP fields

$$a = (a_s, a_e, a_m)$$

- w_m is a 1-bit mantissa
- a_m is an 11-bit mantissa including hidden bit
- Every cycle: 4 weight terms × 4 activations

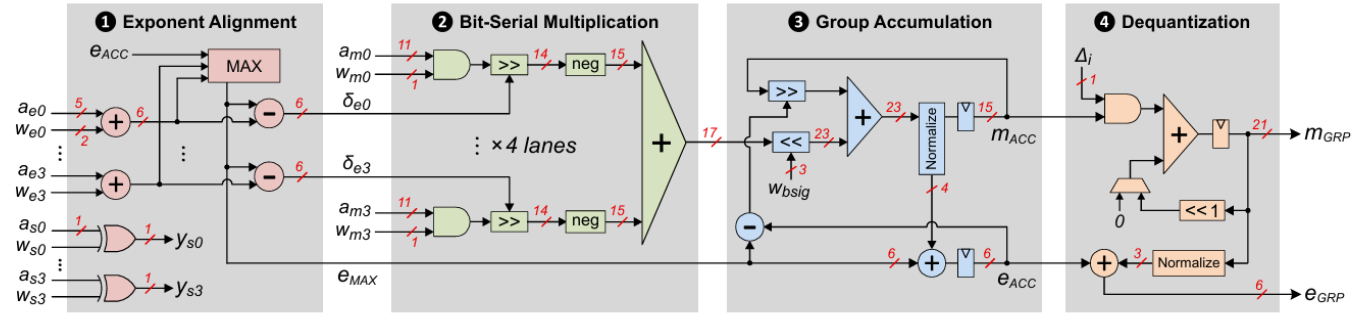


Fig. 5: The microarchitecture of *BitMoD* PE. Every bit-serial weight term contains 1-bit sign (w_s), 2-bit exponent (w_e), 1-bit mantissa (w_m), and a shared bit-significance (w_{bsig}).

Weight Term and FP16 Activation Representation

A key point: BitMoD does not create a new exponent for activations. FP16 activations already contain sign, exponent, and mantissa fields.

Bit-serial weight term from Figure 4

$$w = (w_s, w_e, w_m, w_{bsig})$$

Symbol	Meaning
w_s	sign of weight term
w_e	exponent of term
w_m	1-bit mantissa
w_bsig	bit significance

Generated by INT Booth or FP→Fixed + LOD decoder

FP16 activation fields

$$a = (a_s, a_e, a_m)$$

Sign 1 bit	Exponent 5 bits	Fraction 10 bits

Mantissa used in PE = hidden bit + fraction = 11 bits

PE computes a_e + w_e because FP multiplication adds exponents

Step 1: Exponent Alignment

Step 1 computes the product exponent and sign, then aligns the four products before the dot product is accumulated.

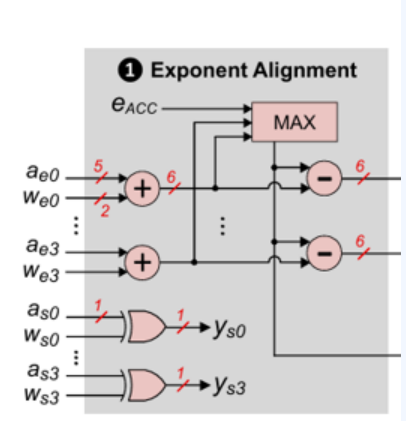
What is computed?

$$A = M_A 2^{E_A},\ W = M_W 2^{E_W}$$

$$A \times W = (M_A M_W) 2^{E_A + E_W}$$

$$e_{product} = a_e + w_e$$

$$y_s = a_s \oplus w_s$$



- Exponent addition follows FP multiplication
- Sign is computed by XOR
- The largest exponent becomes the alignment reference

Detailed example

$$\text{Product exponents} = [5, 3, 4, 5]$$

$$\text{e\_max} = 5$$

$$\delta e_i = e_{max} - e_i$$

$$\boldsymbol{\delta e = [0, 2, 1, 0]}$$

P0	P1	P2	P3
shift 0	shift 2	shift 1	shift 0

δe controls the right shifter in Step 2.

Step 2: Bit-serial Multiplication

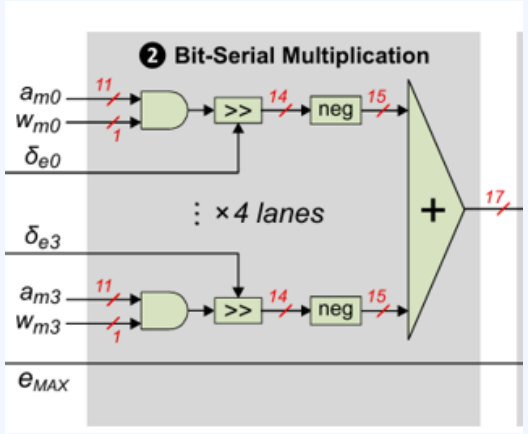
The heavy multiplication is simplified because the weight mantissa in each bit-serial term is only one bit.

1-bit × 11-bit multiplication

$$w_m \times a_m$$

$$w_m \in 0,1$$

w_m	Result
0	0
1	a_m passes through



This replaces a full multiplier with a lightweight bit-serial datapath.

Right shift by δe

Using δe = [0, 2, 1, 0] and mantissas [1.5, 1.25, 1.0, 1.5]

P0	P1	P2	P3
1.5	1.25	1.0	1.5
>>0	>>2	>>1	>>0
1.5	0.3125	0.5	1.5

After alignment, the four products are summed by an adder tree:

$$P = P_0 + P_1 + P_2 + P_3$$

3 extra bits are kept for round-to-nearest-even.

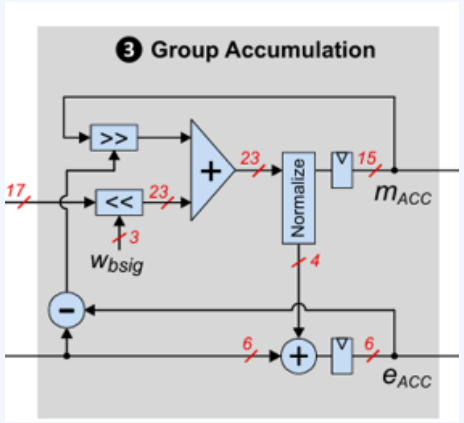
Step 3: Group Accumulation

After one bit-serial dot product is generated, BitMoD applies bit-significance and accumulates terms inside the weight group.

Why accumulation is needed

FP4 value 6 = 4 + 2

Cycle 1: term 4	ACC stores result
Cycle 2: term 2	Add to ACC



Multiple terms are accumulated to reconstruct the original weight contribution.

Detailed numeric example

Dot product result: P = 3

Current bit-significance: $w_{bsig} = 2$

Contribution = $3 \times 2^2 (<<2) = 12$

Previous accumulator: $m_{ACC} = 20$

$m_{ACC} \leftarrow 20 + 12 = 32$

Then normalize m_{ACC} and update e_{ACC}.

Step 4: Bit-serial Dequantization

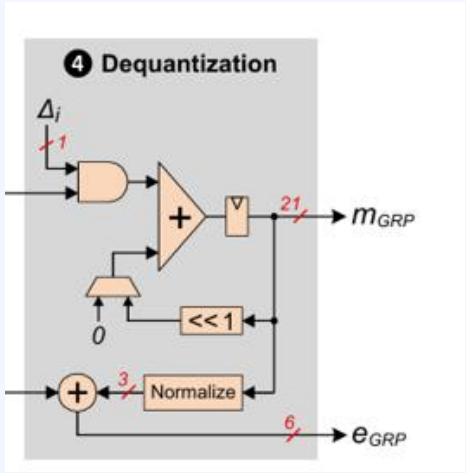
Because BitMoD uses per-group quantization, each group partial sum must be dequantized with its own group scale.

Why INT8 scaling factor matters

- Previous section quantized scaling factors to INT8
- So dequantization no longer needs FP16 scale multiplication
- The PE can process scale bits one by one

Old: $P_g \times \text{FP16}\ \Delta_g$

New: $P_g \times \text{INT8}\ SF_{q,g}$



Detailed example: SF = 13

$$13 = 00001101_2 = 8 + 4 + 1$$

Bit	Action
bit 0 = 1	use m_ACC × 1
bit 1 = 0	skip
bit 2 = 1	use m_ACC << 2
bit 3 = 1	use m_ACC << 3

$$\text{m\_ACC} \times 13 = \text{m\_ACC} + (\text{m\_ACC} << 2) + (\text{m\_ACC} << 3)$$

This is shift-and-add instead of FP multiplication.

Why Dequantization Does Not Stall the PE Pipeline

Although INT8 scale dequantization takes 8 cycles, group dot-product computation takes much longer.

Cycle comparison

Group size G = 128, PE processes 4 weights per cycle

$$128 / 4 = 32$$

$$\text{FP3 has 2 terms} \rightarrow 32 \times 2 = 64 \text{ cycles}$$

$$\text{INT8 dequantization} = 8 \text{ cycles}$$

64 cycles >> 8 cycles, so no pipeline stall.

Term count becomes throughput advantage

Type	Terms	Cycles	Throughput
INT8	4	4	1×
INT6	3	3	1.33×
FP4	2	2	2×
FP3	2	2	2×

Lower weight precision → fewer bit-serial terms → fewer PE cycles

The paper also reports BitMoD PE uses about 24% less area than FP16 PE.

C. BitMoD Accelerator: Overall Architecture

Figure 6 shows how the bit-serial term generator and BitMoD PE are organized into a full accelerator.

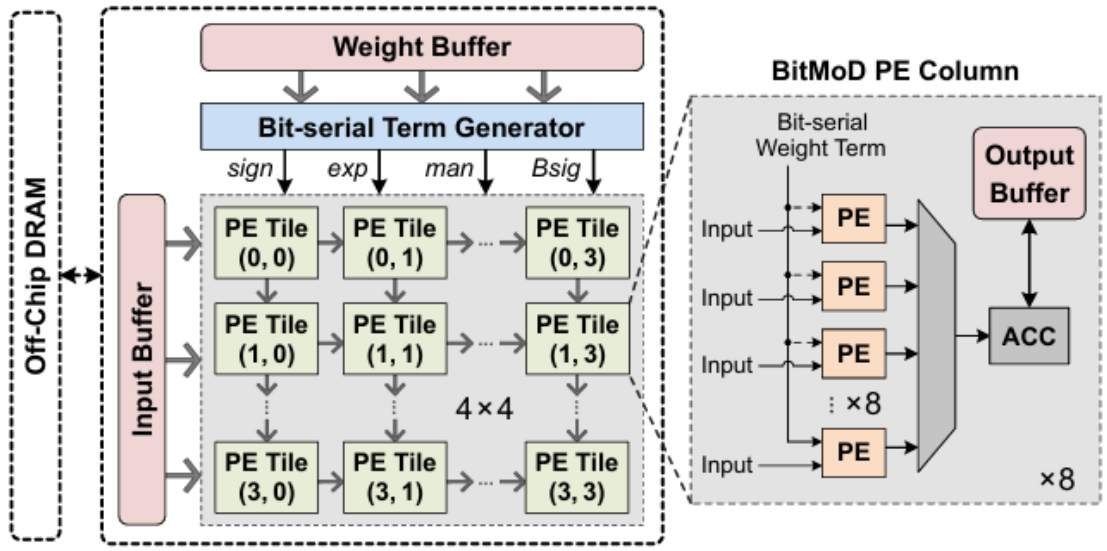


Fig. 6: *BitMoD* accelerator architecture.

Figure 6: banked buffers → term generator → 4×4 PE tiles → PE columns with output buffer + ACC

Main components

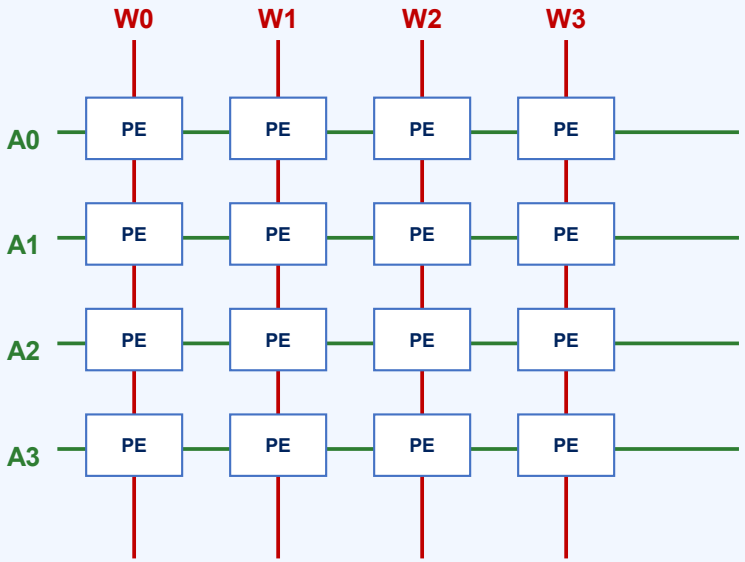
- Banked input and weight buffers provide bandwidth for many PEs.
- Bit-serial Term Generator converts quantized weights into sign / exp / man / Bsig.
- Main PE Array contains 4×4 systolic PE tiles.
- Each PE column has a local output buffer and shared accumulator.

Core goal: exploit PE parallelism while reducing data movement.

Weight-sharing and Input-sharing through Broadcast

BitMoD reuses both weights and inputs by broadcasting them in perpendicular directions across the PE tile.

Broadcast directions



Weight term → column broadcast, Input → row broadcast

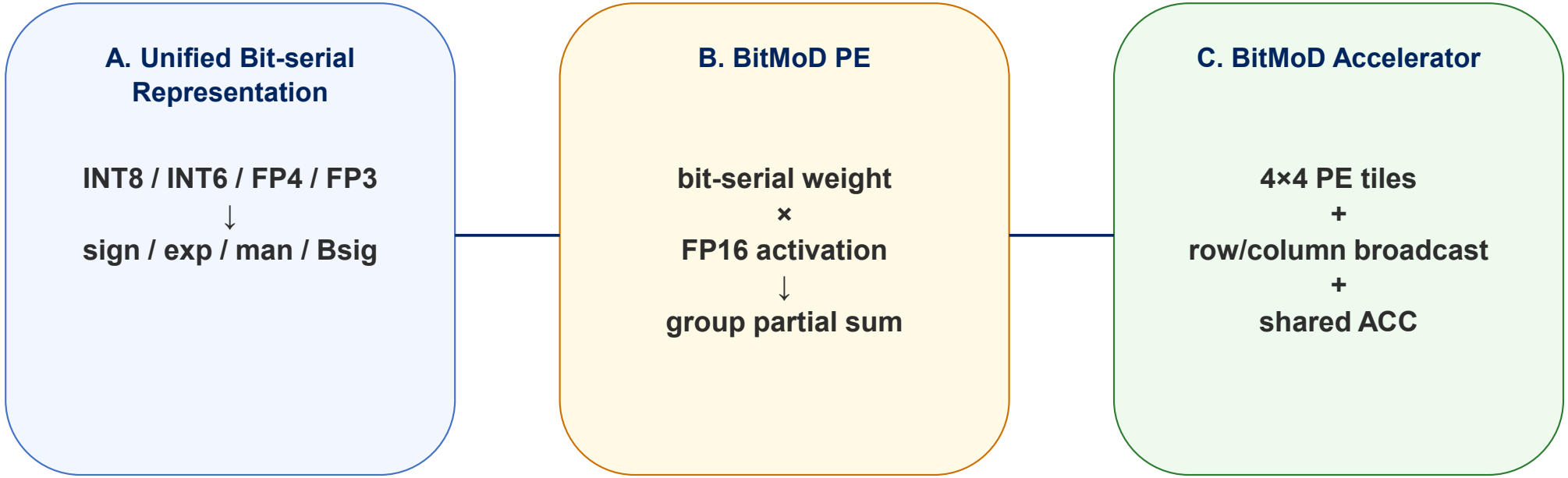
Data reuse effect

- One weight term is fetched once and reused by all PEs in the same column.
- One input activation is fetched once and reused by all PEs in the same row.
- This reduces memory traffic while keeping many PEs active.

Reuse	Broadcast	Benefit
Weight-sharing	Column	less weight fetch
Input-sharing	Row	less input fetch

Hardware Summary: A → B → C

The hardware section connects data type conversion, PE computation, and accelerator-level data reuse into one pipeline.



Final message: BitMoD combines unified bit-serial data representation, mixed-precision PE design, and systolic data reuse to turn low-bit quantization into real accelerator speedup.

Evaluation

Evaluation Setup: What is measured?

BitMoD evaluation checks whether the proposed datatype and hardware actually preserve model quality and improve accelerator efficiency.

Model Accuracy

Low-precision quantization quality
4-bit and 3-bit behavior

Speedup

End-to-end accelerator performance
FP16 / ANT / OliVe comparison

Energy Efficiency

DRAM traffic reduction + compute energy
bit-serial PE advantage

Compatibility

Combination with AWQ, OmniQuant, and SmoothQuant

Experimental methodology

Category	Details
LLMs	OPT-1.3B, Phi-2B, Yi-6B, Llama-2-7B, Llama-2-13B, Llama-3-8B
Discriminative	HellaSwag, WinoGrande, PIQA (zero-shot accuracy)
Generative	Wikitext-2 and C4 (perplexity, PPL)
Baselines	ANT, OliVe, Microscaling(MX), per-group asymmetric INT

Hardware methodology

- BitMoD accelerator is implemented in SystemVerilog RTL.
- Synthesis uses TSMC 28nm technology.
- End-to-end performance is evaluated with a cycle-level simulator.
- All accelerators are compared under the same compute-area budget.

Fair comparison: same compute area, different quantization and hardware designs

Model Accuracy: 4-bit is near-lossless, 3-bit is where BitMoD matters

The paper evaluates both discriminative accuracy and generative perplexity. BitMoD keeps quality especially well when precision gets very low.

< 0.5%

Average accuracy loss
for 4-bit discriminative tasks

< 0.5

Average PPL increase
for 4-bit generative tasks

< 3

Average PPL loss
for 3-bit generative tasks

3-bit

Benefit becomes larger
when quantization is harder

Conventional 3-bit quantization

Quantization error increases
↓
Model quality drops
↓
PPL grows significantly

BitMoD 3-bit quantization

Group-wise special value selection
↓
Range / resolution
↓
Lower quantization error

Takeaway: 4-bit BitMoD is almost FP16-quality; 3-bit BitMoD shows the strongest advantage over ANT, OliVe, MX, and INT3-Asym.

Accelerator Performance: speedup comes from memory and cycles

BitMoD improves speed by reducing weight memory traffic and reducing the number of bit-serial compute cycles.

Why performance improves

- Low-precision weights reduce memory traffic.
- Bit-serial terms reduce PE compute cycles.
- Generative tasks are memory-bound, so they benefit strongly from weight traffic reduction.

Low precision → smaller weights → fewer memory accesses

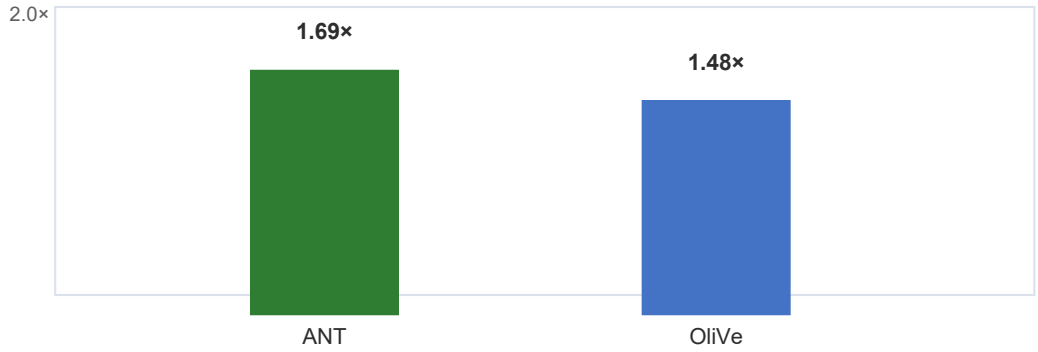
Fewer bit-serial terms → fewer PE cycles

Reported speedup

1.99×
FP16 baseline
Discriminative tasks

2.41×
FP16 baseline
Generative tasks

Average speedup over existing accelerators



BitMoD is faster than ANT and OliVe because it supports per-group low-bit quantization efficiently in hardware.

Energy and Hardware Efficiency

Energy saving mainly comes from fewer DRAM weight accesses and lower compute cost in the bit-serial PE.

2.31×
Energy efficiency
over FP16 baseline

1.48×
Energy efficiency
over ANT

1.31×
Energy efficiency
over OliVe

24%
BitMoD PE area reduction
vs regular FP16 PE

Where energy is saved

Low-precision Weight

↓

DRAM weight traffic decreases

Bit-serial PE

↓

Compute energy decreases

Why bit-serial is flexible

Type	Terms	Cycle effect
INT8	4	baseline
INT6	3	1.33×
FP4	2	2×
FP3	2	2×

Same PE, different cycle count: no separate hardware path for every precision.

Evaluation Summary: What the results prove

The evaluation supports BitMoD as an algorithm-hardware co-design: accurate low-bit quantization plus efficient bit-serial execution.

Item	Reported result
4-bit accuracy	FP16 average accuracy loss < 0.5%
4-bit generative	Average PPL loss < 0.5
3-bit generative	Average PPL loss < 3
FP16 speedup	**1.99× discriminative / 2.41× generative**
ANT / OliVe speedup	**1.69× over ANT / 1.48× over OliVe**
Energy efficiency	**2.31× over FP16 / 1.48× over ANT / 1.31× over OliVe**
Hardware area	BitMoD PE uses about 24% less area than FP16 PE
Compatibility	AWQ, OmniQuant, SmoothQuant can be combined with BitMoD

Q&A