PyTorch Native INT8 Quantization Subclass

Namgyu Youn[1]

[1]AerLabs. namgyu.dev@gmail.com

December 2025

Abstract

This report presents the design and implementation of `Int8Tensor`, a quantized tensor subclass for PyTorch's torchao (ao) library. The primary contribution is *dynamic activation quantization* (W8A8-INT): both weights and activations are represented in INT8, with activation scales computed at runtime, enabling INT8×INT8 matrix multiplication on hardware with native integer arithmetic support. W8A8-INT requires runtime scale computation (activation distributions vary per input) and hardware-specific kernel selection. It integrates via `__torch_dispatch__`, exposes a `Granularity`-based API for per-tensor and per-row quantization, and propagates scales through slice and select for tensor parallelism (PR #3391 [1]).

Keywords: quantization, dynamic activation quantization, PyTorch, tensor subclass

---

1 Introduction

`Int8Tensor` is a quantized tensor subclass in ao that enables INT8 inference. INT8 reduces memory and accelerates inference on INT8-capable hardware.

1.1 Quantization Modes

`Int8Tensor` supports two complementary quantization strategies:

- **Dynamic activation quantization (W8A8-INT).** Both weights and activations are quantized to INT8; scales are computed at runtime since activation distributions vary per batch. This is the primary focus of this report.
- **Weight-only quantization (FP16/BF16×INT8).** Only weights are pre-quantized to INT8; activations remain in floating point. Included for broader hardware compatibility.

2 Technical Foundations

2.1 Asymmetric Integer Quantization

The quantization process is formulated as:

$$Q_x = \left\lfloor \frac{X}{s} + z \right\rceil \tag{1}$$

where X is the floating-point tensor, Q_x is its n-bit quantized counterpart, s is the scaling factor

$$s = \frac{X_{\max} - X_{\min}}{q_{\max} - q_{\min}}, \tag{2}$$

and $z = \lfloor q_{\min} - X_{\min}/s \rceil$ is the zero point. The dequantized approximation is recovered as:

$$\hat{X} = (Q_x - z) \cdot s. \tag{3}$$

2.2 Supported Tensor Operations

Table 1 lists operations dispatched via `__torch_dispatch__`.

Table 1. Tensor operations supported by `Int8Tensor`.

Operation	Description	Limitations
`aten.linear.default`	INT8×INT8 or FP×INT8 matmul	None
`aten.slice.Tensor`	Slicing with scale adjustment	`dim`$\in \{0, 1, 2\}$, `step=1`
`aten.select.int`	Single-index selection	`dim=0` only
`aten.index.Tensor`	Advanced indexing	None

3 Technical Design

3.1 Quantization Granularity

`Int8Tensor` supports per-tensor (single scale) and per-row (one scale per output channel) granularity; per-row is preferred for W8A8-INT. Rather than exposing `block_size` directly — per-row on an $[N, K]$ matrix maps to `block_size=[1, K]`, which is non-obvious — the API takes `Granularity` objects that express intent:

```
from ao.quantization.granularity import PerRow, PerTensor

Int8Tensor.from_hp(weight, granularity=PerRow())
Int8Tensor.from_hp(weight, granularity=PerTensor())
```

Listing 1. High-level granularity API

3.2 Dequantization

The scale shape depends on granularity: scalar for per-tensor, $[N]$ or $[N, 1]$ for per-row. Rather than implementing scale broadcasting manually, `Int8Tensor` delegates to `dequantize_affine`:

```
def dequantize(self, output_dtype=None):

    return dequantize_affine(
        input=self.qdata,
        block_size=self.block_size,
        scale=self.scale,
        output_dtype=output_dtype,
    )
```

Listing 2. Dequantization via ao primitive

3.3 W8A8-INT Linear Kernel

The core challenge in W8A8-INT is computing $Y = XW^\top$ when both X and W are INT8 with separate scale tensors. Converting to float before the matmul eliminates the benefit of INT8 arithmetic. Integer accumulation into INT64 works on CPU but is unsupported by `addmm_cuda` on GPU. The implementation instead uses `int_scaled_matmul`, a ao kernel with CPU and CUDA coverage that performs fused INT8 matmul with scale application:

```
from ao.kernel import int_scaled_matmul

y_dot_scaled = int_scaled_matmul(
    tmp, w_vals_t, x_scales.reshape(-1, 1))
result = y_dot_scaled * w_scales
```

Listing 3. W8A8-INT kernel via int_scaled_matmul

Weight-only path. For FP×INT8, weights are cast to the activation dtype before the matmul, deferring scale multiplication to avoid materialising a full floating-point weight copy:

```
w_vals_t = weight.qdata.t().to(activation.dtype)
m = torch.mm(activation.reshape(-1, activation.shape[-1]), w_vals_t)
result = m * weight.scale
```

Listing 4. Lazy dequantization for weight-only path

3.4 Scale Slicing

Slicing requires consistent scale adjustment across all granularity cases; `_slice_scale` handles this self-contained, without external dependencies:

```
def _slice_scale(scale, data_shape, dim, start, end, step):
    if scale.numel() <= 1:        # Per-tensor: scalar unchanged
        return scale
    if scale.ndim == 1:           # Per-row: slice along dim 0 only
        return (aten.slice.Tensor(scale, 0, start, end, step)
                if dim == 0 else scale)
    # Per-block: map data indices to scale indices
    block_size_for_dim = data_shape[dim] // scale.shape[dim]
    scale_start = start // block_size_for_dim
    scale_end  = (end + block_size_for_dim - 1) // block_size_for_dim
    return aten.slice.Tensor(scale, dim, scale_start, scale_end, 1)
```

Listing 5. Self-contained _slice_scale

4 Design Principles

Three principles guided the design. **Express intent, not implementation:** `Granularity` objects let callers specify *what* quantization is intended without knowing the internal

`block_size` mapping. **Reuse validated primitives:** delegating dequantization to `dequantize_affine` avoids error-prone per-rank scale broadcasting. **Verify hardware constraints early:** `int_scaled_matmul` was chosen specifically for its CPU+CUDA coverage, since generic integer accumulation does not transfer across devices; and `_slice_scale` is kept self-contained to avoid pulling in unrelated subsystems.

5 Benchmarks

Benchmarks run on Qwen3-8B (RTX 5090). Configurations: BF16 baseline, W8A8-INT, and W8A8-INT with `torch.compile` (`INT8+compile`).

5.1 Memory Profile

Table 2. Peak memory — C++ kernel.

Metric	BF16	W8A8-INT	INT8+compile
Allocated (MiB)	15,642.3	11,397.5	8,431.5
Reserved (MiB)	15,658.0	20,678.0	16,086.0
Fragmentation (%)	0.1	44.9	47.6

Table 3. Peak memory — Triton kernel.

Metric	BF16	W8A8-INT	INT8+compile
Allocated (MiB)	15,642.3	8,429.3	8,431.5
Reserved (MiB)	15,658.0	18,302.0	17,274.0
Fragmentation (%)	0.1	53.9	51.2

The Triton backend reduces allocated memory by ≈46% (matching `INT8+compile`), while the C++ backend achieves only ≈27%. All quantized variants show high fragmentation (45–54%) versus near-zero for BF16, attributable to short-lived per-batch activation scale allocations.

Acknowledgements

This work was contributed to ao (PR #3391). The author is especially grateful to Zerry Zhang of the PyTorch core team for invaluable mentorship and technical guidance, and thanks the ao maintainers for their constructive feedback.

References

[1] ao Contributors, "ao: PyTorch-Native Training-to-Serving Model Optimization," GitHub, 2024. https://github.com/pytorch/ao

[2] "QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving;" MLSys'25. https://arxiv.org/abs/2405.04532