PR #36052 完整报告

sgl-project/sglang

[Diffusion] Load self-describing Quanto INT8 encoders

合并时间：2026-08-24 16:37

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/36052

执行摘要

- 一句话：支持自动加载 Quanto INT8 序列化编码器，量化经元数据自发现
- 推荐动作：值得精读，尤其 inspect_quanto_int8_checkpoint 的"自描述元数据 + 前缀集合一致性 + selected 消费校验"模式，以及 text_encoder_loader 中 SRT/diffusion 双线性分派设计；对后续新增序列化量化格式（如其他 weight-only 方案）有直接借鉴价值。阅读时建议先看 base_config.supports_srt_linear_layers 如何做闸门，再看 loader 的查找顺序。

功能与动机

作者在 PR body 中明确目标为 auto-detect self-describing Optimum Quanto qint8 weight-only encoder safetensors from their embedded quantization map，并强调 Serialized checkpoints need only --component-paths.<component>; no quantization flag is required。动机是让公共 DeepBeepMeep MiniMax-H3 编码器检查点开箱即用，避免用户在加载端手工指定量化格式；PR 也特别澄清这是 resident-memory 选项而非 INT8 吞吐声明，说明其价值在于加载灵活性与兼容性而非推理加速。

实现拆解

采用以下 5 步实现（全部位于 python/sglang/multimodal_gen/）：

1. 新增自描述量化契约：新建 runtime/layers/quantization/configs/quanto_int8_config.py，包含 QuantoInt8Config 与 inspect_quanto_int8_checkpoint。后者打开 safetensors 后读取 metadata 中的 quantization_format=quanto 与 quantization_map_base64，base64 + JSON 解码得到 {prefix: {weights, activations}} 映射；校验 map 前缀必须等于所有 .weight._data 张量前缀集合，每个声明层必须是 weight-only qint8（拒绝激活量化），权重张量必须是 2D I8、scale 必须为 [out, 1] 浮点、input/output_scale 必须为标量；随后用 param_name_mapper 把前缀映射到原生模型命名空间并检测映射冲突。QuantoInt8Config.get_quant_method 同时识别 DiffusionLinearBase 与 SrtLinearBase，命中则返回 QuantoInt8LinearMethod 并记录 selected。
2. 新增运行时量化线性：新建 runtime/layers/quantization/quanto_int8.py，QuantoInt8LinearMethod.create_weights 创建 int8 weight 与 per-output 的 weight_scale；apply 把 weight 转成激活 dtype 并与 scale 相乘后走 F.linear，即运行时反量化；normalize_quanto_int8_weights 把 ._data/._scale 张量名还原为原生 weight/weight_scale，并强制 input_scale/output_scale 恒等于 1。

3. 接入 encoder 加载器：runtime/loader/component_loaders/text_encoder_loader.py 中 _get_encoder_quant_config 的查找顺序变为：全局 quant_config → 通用 safetensors metadata → inspect_quanto_int8_checkpoint → Comfy marker 回退；name_mapper 被提前构造并复用。_process_quantized_encoder_weights 与 _require_quantized_encoder_layers 的模块扫描从 LinearBase 扩展为 (LinearBase, SrtLinearBase)，量化方法判定同步排除 SrtUnquantizedLinearMethod，并为 QuantoInt8Config 增加 layer_prefixes 必须全部被 selected 消费的校验；load_model 在 load_weights 前先执行 normalize_quanto_int8_weights。
4. 量化透传到原生视觉塔：base_config.py 给 QuantizationConfig 增加 supports_srt_linear_layers=False 默认字段，QuantoInt8Config 置为 True；qwen3vl.py 仅在 quant_config.supports_srt_linear_layers 为真时把 quant_config 传给 Qwen3VLVisionTransformer；qwen3vl_vision.py / qwen_vl_vision.py 的 Transformer、Block、Attention 均新增可选 quant_config 参数并透传给 qkv_proj / proj / MLP；minimax_h3_qwen3vl.py 补充 language_model 前缀映射以匹配公共检查点命名。
5. 测试与文档：新增 test_quanto_int8.py（端到端驱动 ReplicatedLinear、拒绝激活量化、辅助 scale 校验），扩展 test_qwen3vl_vision.py（验证 SRT 视觉塔接收 QuantoInt8Config 后参数为 int8 且全部前缀被选中），调整 test_text_encoder_loader.py；docs/cookbook/diffusion/MiniMax/MiniMax-H3.mdx 与 docs/docs/sglang-diffusion/quantization.mdx 补充 Quanto 用法说明。

关键文件：

- python/sglang/multimodal_gen/runtime/layers/quantization/configs/quanto_int8_config.py（模块 量化契约；类别 source；类型 dependency-wiring；符号 QuantoInt8Config, get_quant_method, inspect_quanto_int8_checkpoint, from_config）：新增 QuantoInt8Config 与 inspect_quanto_int8_checkpoint，是自描述 Quanto 检查点准入与分派的核心契约。
- python/sglang/multimodal_gen/runtime/layers/quantization/quanto_int8.py（模块 量化层；类别 source；类型 dependency-wiring；符号 QuantoInt8LinearMethod, create_weights, apply, normalize_quanto_int8_weights）：QuantoInt8LinearMethod 是运行时执行单元，负责 int8 权重创建与反量化 forward；normalize_quanto_int8_weights 解决张量名映射。
- python/sglang/multimodal_gen/runtime/loader/component_loaders/text_encoder_loader.py（模块 加载器；类别 source；类型 dependency-wiring；符号 _get_encoder_quant_config, _require_quantized_encoder_layers, _process_quantized_encoder_weights）：把 Quanto 检测接入现有 encoder 量化加载流程，并扩展 SRT linear 支持与“全部消费”校验，是加载链路的枢纽。
- python/sglang/multimodal_gen/runtime/models/encoders/qwen3vl_vision.py（模块 视觉塔；类别 source；类型 data-contract；符号 Qwen3VLVisionBlock.init, Qwen3VLVisionTransformer.init）：让视觉塔可接收量化配置并按前缀构造块，是 Quanto 量化进入原生 Qwen3-VL 视觉塔的桥梁。
- python/sglang/multimodal_gen/runtime/models/encoders/qwen3vl.py（模块 视觉塔；类别 source；类型 data-contract；符号 Qwen3VLForConditionalGeneration.init）：决定

视觉塔是否接收量化配置的闸门，依赖 supports_srt_linear_layers，避免影响其他量化路径。

- python/sglang/multimodal_gen/runtime/models/encoders/qwen_vl_vision.py（模块 视觉塔；类别 source；类型 data-contract；符号 QwenVLVisionAttention.init）：通用 QwenVL 视觉注意力层获得 quant_config，使视觉塔内 qkv_proj/proj 可参与量化分派。
- python/sglang/multimodal_gen/runtime/layers/quantization/configs/base_config.py（模块 基础契约；类别 source；类型 core-logic；符号 QuantizationConfig.supports_srt_linear_layers）：给 QuantizationConfig 增加 supports_srt_linear_layers 标志，是 Quant 配置能否作用于 SRT 线性层的总开关。
- python/sglang/multimodal_gen/runtime/models/encoders/minimax_h3_qwen3vl.py（模块 视觉塔；类别 source；类型 data-contract；符号 _PARAM_NAMES_MAPPING）：参数名映射补上 language_model 前缀，适配公共 DeepBeepMeep 检查点的命名。
- python/sglang/multimodal_gen/test/unit/test_quanto_int8.py（模块 单元测试；类别 test；类型 test-coverage；符号 _save_quanto_checkpoint, test_quanto_checkpoint_drives_native_linear_end_to_end, test_quanto_checkpoint_rejects_activation_quantization, test_quanto_weight_only_auxiliary_scales_must_be_identity）：新增端到端契约测试，覆盖检查点驱动、拒绝激活量化、辅助 scale 恒 1。
- python/sglang/multimodal_gen/test/unit/test_qwen3vl_vision.py（模块 单元测试；类别 test；类型 test-coverage；符号 test_native_vision_accepts_srt_linear_quantization）：验证 SRT 视觉塔在 QuantoInt8Config 下把所有声明层切到 int8，守护新的数据契约。
- docs/cookbook/diffusion/MiniMax/MiniMax-H3.mdx（模块 文档；类别 docs；类型 documentation）：补充 MiniMax-H3 使用 Quanto INT8 序列化 encoder 的组件覆盖说明。

关键符号：QuantoInt8Config.get_quant_method, inspect_quanto_int8_checkpoint, QuantoInt8LinearMethod.create_weights, QuantoInt8LinearMethod.apply, normalize_quanto_int8_weights, _get_encoder_quant_config, _require_quantized_encoder_layers, Qwen3VLVisionTransformer.init, QwenVLVisionAttention.init

关键源码片段

python/sglang/multimodal_gen/runtime/layers/quantization/configs/quanto_int8_config.py

新增 QuantoInt8Config 与 inspect_quanto_int8_checkpoint，是自描述 Quanto 检查点准入与分派的核心契约。

```
# SPDX-License-Identifier: Apache-2.0
# Config 与 checkpoint 准入逻辑：从 Optimum Quanto qint8 safetensors 元数据自描述发现量化。

from __future__ import annotations

import base64
import json
```

```python
from collections.abc import Callable
from typing import Any

import torch
from safetensors import safe_open

from sglang.multimodal_gen.runtime.layers.linear import (
    LinearBase as DiffusionLinearBase,
    UnquantizedLinearMethod as DiffusionUnquantizedLinearMethod,
)
from sglang.multimodal_gen.runtime.layers.quantization.configs.base_config import (
    QuantizationConfig,
    QuantizeMethodBase,
)
from sglang.multimodal_gen.runtime.layers.quantization.quanto_int8 import (
    QuantoInt8LinearMethod,
)
from sglang.srt.layers.linear import LinearBase as SrtLinearBase
from sglang.srt.layers.quantization.unquant import (
    UnquantizedLinearMethod as SrtUnquantizedLinearMethod,
)

_FLOAT_DTYPES = {"BF16", "F16", "F32"}


class QuantoInt8Config(QuantizationConfig):
    """把 quantization map 中声明为 qint8 的线性层分派到 QuantoInt8LinearMethod。

    与 Comfy FP8/Kitchen 方案不同，该配置完全由检查点自带元数据驱动，
    不能从 model config 文件构造，因此 from_config 直接拒绝。
    """

    # 允许 SRT 线性层（如 Qwen3-VL 视觉塔）参与分派，这是本 PR 的关键开关
    supports_srt_linear_layers = True

    def __init__(self, layer_prefixes: set[str]) -> None:
        super().__init__()
        self.layer_prefixes = layer_prefixes # 检查点声明的全部量化层前缀
        self.selected: set[str] = set() # 运行时实际命中并量化的前缀

    @classmethod
    def get_name(cls) -> str:
        return "quanto_int8"

    @classmethod
    def get_supported_act_dtypes(cls) -> list[torch.dtype]:
        # weight-only 方案，激活仅支持 BF16/FP16
        return [torch.bfloat16, torch.float16]
```

```
    @classmethod
    def get_min_capability(cls) -> int:
        # 不依赖特定 GPU 算力，因为执行时反量化回浮点
        return 0

    @staticmethod
    def get_config_filenames() -> list[str]:
        # 不读取独立配置文件，全部来自 safetensors metadata
        return []

    @classmethod
    def from_config(cls, config: dict[str, Any]) -> QuantoInt8Config:
        # 该配置无法从 JSON config 构造，必须由检查点元数据驱动
        raise ValueError(
            "QuantoInt8Config must be constructed from safetensors metadata"
        )

    def get_quant_method(
        self, layer: torch.nn.Module, prefix: str
    ) -> QuantizeMethodBase | None:
        # 同时支持 diffusion 原生线性层与复用 SRT 的线性层
        if isinstance(layer, DiffusionLinearBase):
            unquantized_method = DiffusionUnquantizedLinearMethod
        elif isinstance(layer, SrtLinearBase):
            unquantized_method = SrtUnquantizedLinearMethod
        else:
            return None
        if prefix not in self.layer_prefixes:
            # 未声明量化的层保持非量化
            return unquantized_method()
        # 记录命中，供后续"全部消费"校验
        self.selected.add(prefix)
        return QuantoInt8LinearMethod()
```

python/sglang/multimodal_gen/runtime/layers/quantization/quanto_int8.py

QuantoInt8LinearMethod 是运行时执行单元，负责 int8 权重创建与反量化 forward；normalize_quanto_int8_weights 解决张量名映射。

```
# SPDX-License-Identifier: Apache-2.0
# Runtime 操作：保持 qint8 权重打包存储，仅对激活矩阵反量化后执行浮点线性计算。

from __future__ import annotations

from collections.abc import Iterable, Iterator
from typing import Any

import torch
import torch.nn.functional as F
from torch.nn.parameter import Parameter
```

```
from sglang.multimodal_gen.runtime.layers.linear import LinearMethodBase
from sglang.multimodal_gen.runtime.utils.weight_attrs import set_weight_attrs


class QuantoInt8LinearMethod(LinearMethodBase):
    """Keep qint8 weights packed and dequantize only the active matrix."""

    def create_weights(
        self,
        layer: torch.nn.Module,
        input_size_per_partition: int,
        output_partition_sizes: list[int],
        input_size: int,
        output_size: int,
        params_dtype: torch.dtype,
        **extra_weight_attrs: Any,
    ) -> None:
        # int8 权重不参与梯度
        weight = Parameter(
            torch.empty(
                sum(output_partition_sizes),
                input_size_per_partition,
                dtype=torch.int8,
            ),
            requires_grad=False,
        )
        set_weight_attrs(weight, {"input_dim": 1, "output_dim": 0})
        set_weight_attrs(weight, extra_weight_attrs)
        layer.register_parameter("weight", weight)

        # per-output 反量化 scale，形状为 [out, 1]
        weight_scale = Parameter(
            torch.empty(
                sum(output_partition_sizes),
                1,
                dtype=params_dtype,
            ),
            requires_grad=False,
        )
        set_weight_attrs(weight_scale, {"output_dim": 0})
        set_weight_attrs(weight_scale, extra_weight_attrs)
        layer.register_parameter("weight_scale", weight_scale)

    def apply(
        self,
        layer: torch.nn.Module,
        x: torch.Tensor,
        bias: torch.Tensor | None = None,
    ) -> torch.Tensor:
```

```
        # 每次前向都在显存中展开为浮点矩阵，因此这是 resident-memory
        # 方案而非 INT8 吞吐方案
        weight = layer.weight.to(dtype=x.dtype)
        weight.mul_(layer.weight_scale.to(dtype=x.dtype))
        return F.linear(x, weight, bias)


def normalize_quanto_int8_weights(
    weights: Iterable[tuple[str, torch.Tensor]],
) -> Iterator[tuple[str, torch.Tensor]]:
    """把扁平化 Quanto 张量名映射为原生 linear 参数名。"""

    for name, tensor in weights:
        if name.endswith((".input_scale", ".output_scale")):
            # weight-only 模式下输入 / 输出辅助 scale 必须恒为 1，否则直接拒绝
            if tensor.numel() != 1 or tensor.item() != 1:
                raise ValueError(f"Quanto weight-only scale {name!r} must equal 1")
            continue
        if name.endswith(".weight._data"):
            # 量化数据张量直接映射回原生 weight
            name = name.removesuffix("._data")
        elif name.endswith(".weight._scale"):
            # scale 张量映射到原生 weight_scale
            name = name.removesuffix("._scale") + "_scale"
        yield name, tensor
```

python/sglang/multimodal_gen/runtime/loader/component_loaders/text_encoder_loader.py

把 Quanto 检测接入现有 encoder 量化加载流程，并扩展 SRT linear 支持与“全部消费”校验，是加载链路的枢纽。

```
# 编码器量化配置查找：显式配置 → 通用 safetensors metadata → Quanto → Comfy 回退
def _get_encoder_quant_config(
    component_config: dict,
    component_model_path: str,
    component_weights_path: str,
    model_cls: type[nn.Module] | None = None,
):
    quant_config = get_quant_config(component_config, component_model_path)

    # 提前构造名字映射器：Quanto 与 Comfy 的额外前缀都需要映射到原生参数名
    name_mapper = None
    if model_cls is not None:
        mapping = vars(model_cls).get("param_names_mapping", {})
        if mapping:
            mapping_fn = get_param_names_mapping(mapping)

            def name_mapper(name: str) -> str:
                # 前缀映射基于 name.weight 生成，因此补回 .weight 再剥离
```

```
            mapped_name, merge_index, _ = mapping_fn(f"{name}.weight")
            if merge_index is not None:
                # 融合 /stacked 参数无法用逐层量化前缀描述，直接拒绝
                raise ValueError(
                    "Serialized quantized component weights cannot use a "
                    "stacked parameter-name mapping"
                )
            return mapped_name.removesuffix(".weight")

    if (
        quant_config is None
        and component_weights_path != component_model_path
        and component_weights_path.endswith(".safetensors")
    ):
        # 通用通道：直接读 safetensors metadata 中的量化信息
        quant_config = get_quant_config_from_safetensors_metadata(
            component_weights_path
        )

    if quant_config is None and component_weights_path.endswith(".safetensors"):
        # 新通道：Quanto 自描述检查点
        quant_config = inspect_quanto_int8_checkpoint(
            component_weights_path,
            param_name_mapper=name_mapper,
        )
        if quant_config is None:
            # 回退：Comfy 量化 marker
            markers = inspect_comfy_quant_markers(
                [component_weights_path],
                param_name_mapper=name_mapper,
            )
            quant_config = resolve_comfy_checkpoint_quantization(markers)
    return quant_config


# 校验“检查点声明的所有量化层都必须在模型里被实际消费”
if isinstance(quant_config, (ComfyFp8Config, KitchenInt8Config, KitchenW4A8Config)):
    expected = set(quant_config.layer_markers)
    selected = set(quant_config.selected)
elif isinstance(quant_config, QuantoInt8Config):
    expected = quant_config.layer_prefixes
    selected = quant_config.selected
else:
    expected = set()
    selected = set()
if expected:
    missing = expected - selected
    if missing:
        raise ComponentCheckpointUnsupportedError(
```

```
            f"The native {type(model).__name__} implementation did not consume "
            f"serialized quantization markers for {component_name!r}: "
            f"{sorted(missing)[:5]}"
        )
```

评论区精华

该 PR 没有实质性 review 讨论（review_comments_count 为 0），仅有一条 mintlify 机器人对文档 preview 的机器评论，合并由作者自行完成。PR 中值得注意的设计声明都写在 body 里：作者明确 Each active qint8 matrix is dequantized for BF16/FP16 linear math, so this is a resident-memory option rather than an INT8 throughput claim，即团队共识是把它当成加载兼容性方案，而非 INT8 推理加速方案。

- 暂无高价值评论线程

风险与影响

- 风险： 1) 回归面：text_encoder_loader 的量化层扫描从 DiffusionLinearBase 扩展到 (LinearBase, SrtLinearBase)，所有复用 SRT 线性层的 diffusion 编码器都会受影响，目前靠 QuantizationConfig.supports_srt_linear_layers 闸门隔离；若未来其他量化配置漏置该标志，可能被静默跳过或误进视觉塔。 2) 性能与显存：QuantoInt8LinearMethod.apply 每个前向都把 int8 权重转成浮点并乘 scale，整矩阵反量化常驻显存，PR body 已声明这是 resident-memory 选项而非 INT8 吞吐收益；对低显存场景需关注峰值内存。 3) 契约严格性：inspect_quanto_int8_checkpoint 强制 metadata 中 quantization_format=quanto、map 与张量前缀完全一致、激活量化不支持、辅助 scale 必须恒 1；公共检查点在格式细节上若有偏差会直接加载失败（fail-fast，可接受但需监控）。 4) CI 状态：PR 自述 local tests were intentionally not run，且 PR Test (Extra) 显示为失败、AMD 运行未完成，合并前未看到失败原因澄清。
  - 影响：用户侧：MiniMax-H3 / Qwen3-VL 编码器用户可零额外 flag 加载 Quanto qint8 序列化检查点，公共 DeepBeepMeep MiniMax-H3 encoder（350 语言 + 108 视觉 qint8 线性层）可直接通过 --component-paths 使用。系统侧：diffusion 量化体系从 Comfy FP8/W4A8 扩展到 Optimum Quanto，并且首次打通“复用 SRT 线性层参与 diffusion 量化分派”的通道。团队侧：新增一类检查点契约需要维护，配套测试覆盖了契约校验、数值正确性和消费校验；影响面集中在 multimodal_gen，对 SRT 主推理路径基本无侵入。
  - 风险标记：加载流程改动，CI Extra 未通过，运行时反量化无吞吐收益，严格元数据契约，本地测试未跑

关联脉络

- PR #36036 [Diffusion] Load serialized Comfy W4A8 checkpoints: 同一条“safetensors 元数据自述量化”加载链路，均从检查点元数据自动发现 quant_config，本 PR 的加载查找顺序直接受其影响。
- PR #36060 [Diffusion] Infer Comfy FP8 activation scaling: 同为序列化量化加载改造，涉及 quantization_utils / 加载器，与本 PR 构成 diffusion 序列化量化能力矩阵。

- PR #36063 [Diffusion] Reuse SRT quantization contracts and MXFP8 kernels: 与 base_config 的量化契约复用方向一致，本 PR 的 supports_srt_linear_layers 标志延续了这一趋势。
- PR #36067 [Diffusion] Load Diffusers MiniMax H3 components natively: MiniMax-H3 原生加载系列，本 PR 的参数映射与视觉塔改造依赖其 minimax_h3 结构。
- PR #36078 [Diffusion] Add composable component weight path CLI: 提供 --component-paths 入口，是本 PR 描述的序列化检查点唯一必需用户参数。
- PR #36076 [Diffusion] Support compact Qwen3-VL conditioning for MiniMax H3: 与本 PR 都改动 Qwen3VL 视觉塔与 MiniMax-H3 encoder，共同完善 MiniMax-H3 编码链路。