PR #44260 完整报告

vllm-project/vllm

Add the QuantizedActivation linear-kernel contract

合并时间：2026-06-13 04:48

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/44260

执行摘要

- 一句话：引入 QuantizedActivation 契约，融合 kernel 可跳过线性层输入量化
- 推荐动作：值得精读，尤其是 quant_activation.py 的契约设计和 as_quantized_activation 的防御式验证模式。这是 vllm 量化融合架构迁移的核心基石，理解它有助于参与后续手动融合工作。

功能与动机

Issue #43224 推动将 torch.compile 融合迁移为手动融合。本 PR 实现消费者侧契约，使融合 kernel 能直接传递预量化激活给线性层，避免重复量化，为后续 norm+quant 等融合铺路。

实现拆解

1. 新增 vllm/model_executor/layers/fusion/quant_activation.py，定义 QuantizedActivation 数据类（data, scale, orig_dtype, orig_shape, quant_key），以及 expose_input_quant_key（将 kernel 声明的 key 设置到 layer.input_quant_key）和 as_quantized_activation（验证并拆包预量化激活或返回 None）两个函数。
2. 在 MMLinearLayerConfig 基类和 ScaledMMLinearKernel 基类中添加 input_quant_key() 默认返回 None 的方法，子类可覆盖声明支持的 QuantKey。
3. 在 FlashInferCutlassNvFp4LinearKernel 和 CutlassFP8ScaledMMLinearKernel 中覆盖 input_quant_key() 返回对应 key，FlashInferFP8ScaledMMLinearKernel 同样覆盖；其余 kernel 返回 None 维持原行为。
4. 在 FP8ScaledMMLinearKernel.apply_weights 和 FlashInferCutlassNvFp4LinearKernel.apply_weights 中添加入口检测：调用 as_quantized_activation(x) 判断输入是否为 QuantizedActivation，若是则直接使用预量化 data/scale 并恢复原始 shape/dtype，否则执行原有的量化流程。
5. 新增 tests/fusion/test_quant_activation_contract.py，包含四个测试：测试只有声明的三个 kernel 返回非 None key；测试这些 kernel 的 apply_weights 中实际调用 as_quantized_activation；测试 expose_input_quant_key 正确标记 layer；测试 as_quantized_activation 在 key 不匹配时抛出 AssertionError。
6. 在 .buildkite/test_areas/quantization.yaml 中添加新测试文件的执行路径，确保 CI 覆盖。

关键文件：

- vllm/model_executor/layers/fusion/quant_activation.py（模块 融合层；类别 source；类型 data-contract；符号 QuantizedActivation, expose_input_quant_key, as_quantized_activation）：核心契约文件，定义 QuantizedActivation 数据类和两个辅助函数，消费者侧契约入口。
- tests/fusion/test_quant_activation_contract.py（模块 测试契约；类别 test；类型 test-coverage；符号 _all_kernel_classes, _probe, _resolved_apply_weights, test_only_known_backends_support_prequantized_input）：契约测试文件，验证所有已知 kernel 的 input_quant_key 声明和 apply_weights 中 pre-quantized 消费。
- vllm/model_executor/kernels/linear/scaled_mm/ScaledMMLinearKernel.py（模块 线性 kernel；类别 source；类型 data-contract；符号 input_quant_key）：展示了 input_quant_key 基类方法和 FP8ScaledMMLinearKernel.apply_weights 的 QuantizedActivation 消费模式。
- vllm/model_executor/kernels/linear/nvfp4/flashinfer.py（模块 线性 kernel；类别 source；类型 data-contract；符号 input_quant_key）：FlashInferCutlassNvFp4LinearKernel 覆盖 input_quant_key 返回 kNvfp4Dynamic，并调整 apply_weights 支持 QuantizedActivation。
- vllm/model_executor/kernels/linear/base.py（模块 线性 kernel；类别 source；类型 data-contract；符号 input_quant_key）：在 MMLinearLayerConfig 基类中添加 input_quant_key 默认实现。
- .buildkite/test_areas/quantization.yaml（模块 CI 配置；类别 config；类型 configuration）：在 CI 测试区域配置中添加融合契约测试路径。

关键符号：QuantizedActivation, expose_input_quant_key, as_quantized_activation, ScaledMMLinearKernel.input_quant_key, FP8ScaledMMLinearKernel.apply_weights, FlashInferCutlassNvFp4LinearKernel.input_quant_key, FlashInferCutlassNvFp4LinearKernel.apply_weights

关键源码片段

vllm/model_executor/layers/fusion/quant_activation.py

核心契约文件，定义 QuantizedActivation 数据类和两个辅助函数，消费者侧契约入口。

```
# SPDX-License-Identifier: Apache-2.0
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project
"""
QuantizedActivation 是融合 kernel 产生的预量化激活，
线性层可以直接消费，跳过自身的输入量化步骤。
"""

from dataclasses import dataclass

import torch

from vllm.model_executor.layers.quantization.utils.quant_utils import QuantKey

@dataclass
```

```
class QuantizedActivation:
    """预量化激活及其缩放因子和原始元数据。

    quant_key 描述数据格式（dtype、scale 粒度、值打包）。
    layout/padding 等细节需遵循 kernel 约定。
    TODO(mgoin): 将 layout 和 padding 要求编码进契约。
    """
    data: torch.Tensor # 量化后的数据
    scale: torch.Tensor # 缩放因子
    orig_dtype: torch.dtype # 原始的浮点 dtype
    orig_shape: torch.Size # 原始 shape
    quant_key: QuantKey # 量化键，标识量化方案


def expose_input_quant_key(layer: torch.nn.Module, kernel) -> None:
    """将 kernel 支持的预量化 key 暴露到 layer.input_quant_key 属性。

    如果 kernel.input_quant_key() 返回 None，则不设置属性，
    从而非支持后端不会收到 QuantizedActivation。

    TODO(mgoin): 生产者也需要消费方的量化 scale（如 static input scale、global scale），
    此处后续应暴露这些信息，避免生产者访问 kernel 特有属性。
    """
    key = kernel.input_quant_key()
    if key is not None:
        layer.input_quant_key = key


def as_quantized_activation(
    x: "torch.Tensor | QuantizedActivation", expected_key: QuantKey | None
) -> "QuantizedActivation | None":
    """验证并拆包预量化激活。

    如果 x 是 QuantizedActivation 且 quant_key 匹配 expected_key，返回它；
    如果 x 是普通 tensor，返回 None（调用方自行量化）；
    如果 key 不匹配，触发断言，防止静默错误。
    """
    if not isinstance(x, QuantizedActivation):
        return None
    assert x.quant_key == expected_key, (
        f"QuantizedActivation key {x.quant_key} != consumer kernel "
        f"input_quant_key {expected_key}"
    )
    return x
```

tests/fusion/test_quant_activation_contract.py

契约测试文件，验证所有已知 kernel 的 input_quant_key 声明和 apply_weights 中 pre-quantized 消费。

```
# SPDX-License-Identifier: Apache-2.0
"""Contract tests for the QuantizedActivation linear-kernel integration."""

import pytest
import torch

from vllm.model_executor.kernels.linear import (
    _POSSIBLE_FP8_BLOCK_KERNELS, _POSSIBLE_FP8_KERNELS,
    _POSSIBLE_INT8_KERNELS, _POSSIBLE_NVFP4_KERNELS,
)
from vllm.model_executor.kernels.linear.nvfp4.base import NvFp4LinearKernel,
NvFp4LinearLayerConfig
from vllm.model_executor.kernels.linear.nvfp4.flashinfer import (
    FlashInferCutlassNvFp4LinearKernel, FlashInferTrtllmNvFp4LinearKernel,
)
from vllm.model_executor.kernels.linear.scaled_mm.cutlass import
CutlassFP8ScaledMMLinearKernel
from vllm.model_executor.kernels.linear.scaled_mm.flashinfer import
FlashInferFP8ScaledMMLinearKernel
from vllm.model_executor.kernels.linear.scaled_mm.ScaledMMLinearKernel import (
    FP8ScaledMMLinearLayerConfig, Int8ScaledMMLinearKernel,
    Int8ScaledMMLinearLayerConfig,
)
from vllm.model_executor.layers.fusion.quant_activation import (
    QuantizedActivation, as_quantized_activation, expose_input_quant_key,
)
from vllm.model_executor.layers.quantization.utils.quant_utils import kFp8StaticTensorSym,
kNvfp4Dynamic
from vllm.platforms import current_platform

# 唯一支持预量化的三个 kernel
SUPPORTING = {
    CutlassFP8ScaledMMLinearKernel,
    FlashInferFP8ScaledMMLinearKernel,
    FlashInferCutlassNvFp4LinearKernel,
}

def _all_kernel_classes() -> list[type]:
    """汇总所有已注册的线性 kernel 类。"""
    seen: dict[type, None] = {}
    for registry in (_POSSIBLE_FP8_KERNELS, _POSSIBLE_FP8_BLOCK_KERNELS,
                     _POSSIBLE_INT8_KERNELS, _POSSIBLE_NVFP4_KERNELS):
        for kernels in registry.values():
            for cls in kernels:
                seen.setdefault(cls, None)
    return list(seen)

def _probe(cls: type):
    """创建一个轻量 kernel 实例（跳过 __init__），以便查询 input_quant_key。"""
```

```
    obj = cls.__new__(cls)
    if issubclass(cls, NvFp4LinearKernel):
        obj.config = NvFp4LinearLayerConfig()
    elif issubclass(cls, Int8ScaledMMLinearKernel):
        obj.config = Int8ScaledMMLinearLayerConfig(
            is_static_input_scheme=True, is_channelwise=False, input_symmetric=True)
    else:
        obj.config = FP8ScaledMMLinearLayerConfig(
            weight_quant_key=kFp8StaticTensorSym, activation_quant_key=kFp8StaticTensorSym,
            weight_shape=(16,16), input_dtype=torch.bfloat16, out_dtype=torch.bfloat16)
    return obj

def test_only_known_backends_support_prequantized_input():
    """确保只有 SUPPORTING 中的 kernel 返回非 None 的 input_quant_key。"""
    declarers = {c for c in _all_kernel_classes() if _probe(c).input_quant_key()}
    assert declarers == SUPPORTING

def test_supporting_backend_declares_consume_via_helper():
    """确保 SUPPORTING 中的 kernel 的 apply_weights 调用了 as_quantized_activation。"""
    for cls in SUPPORTING:
        fn = _resolved_apply_weights(cls)
        assert "as_quantized_activation" in fn.__code__.co_names, cls.__name__

def test_as_quantized_activation_validates_key():
    """确保 as_quantized_activation 在 key 不匹配时抛出 AssertionError。"""
    qa = QuantizedActivation(
        data=torch.zeros(2, 4, dtype=current_platform.fp8_dtype()),
        scale=torch.tensor(1.0),
        orig_dtype=torch.bfloat16,
        orig_shape=torch.Size([2, 4]),
        quant_key=kFp8StaticTensorSym,
    )
    with pytest.raises(AssertionError):
        as_quantized_activation(qa, kNvfp4Dynamic)
    with pytest.raises(AssertionError):
        as_quantized_activation(qa, None)
    # 普通 tensor 应返回 None
    assert as_quantized_activation(torch.zeros(2, 4), kFp8StaticTensorSym) is None
    # 匹配的 QuantizedActivation 应返回自身
    assert as_quantized_activation(qa, kFp8StaticTensorSym) is qa
```

vllm/model_executor/kernels/linear/scaled_mm/ScaledMMLinearKernel.py

展示了 input_quant_key 基类方法和 FP8ScaledMMLinearKernel.apply_weights 的 QuantizedActivation 消费模式。

```
class ScaledMMLinearKernel(Generic[_ConfigT, _ParamsT], ABC):
    # ... 构造函数、抽象方法等 ...

    def input_quant_key(self) -> QuantKey | None:
```

```
        """返回本 kernel 支持消费的预量化激活 QuantKey。

        手动融合据此决定是否将量化外提到上游融合 kernel。
        返回 None 表示 kernel 需要自行量化（有自定义 padding 或 swizzling 等）。
        若返回非 None，apply_weights 必须通过 as_quantized_activation 消费输入。
        """
        return None


class FP8ScaledMMLinearKernel(ScaledMMLinearKernel[FP8ScaledMMLinearLayerConfig, _
FP8ParamsT], ABC):
    def apply_weights(
        self,
        layer: torch.nn.Module,
        x: torch.Tensor | QuantizedActivation,
        bias: torch.Tensor | None = None,
    ) -> torch.Tensor:
        w, w_s, x_s, x_s_ub = self._get_layer_params(layer)

        # 尝试将输入作为预量化激活拆包
        qa = as_quantized_activation(x, self.input_quant_key())
        if qa is not None:
            # 预量化路径：直接使用 data 和 scale，恢复原始 shape/dtype
            x_data, x_s = qa.data, qa.scale
            orig_shape, orig_dtype = qa.orig_shape, qa.orig_dtype
            assert x_data.dtype == fp8_dtype
        else:
            # 普通 tensor 路径：执行内部量化
            assert isinstance(x, torch.Tensor)
            x_data = x
            orig_shape, orig_dtype = x.shape, x.dtype

        x_2d = x_data.view(-1, x_data.shape[-1])
        output_shape = [*orig_shape[:-1], w.shape[1]]
        out_dtype = orig_dtype if self.config.out_dtype is None else self.config.out_dtype

        # 只有在没有预量化时才执行量化
        if qa is None:
            x_2d, x_s = self.quant_fp8(x_2d, x_s, x_s_ub)

        return self.apply_scaled_mm(
            A=x_2d, B=w, out_dtype=out_dtype, As=x_s, Bs=w_s, ...
        )
```

评论区精华

BadrBasowid 建议提取 apply_mm 静态方法减少 apply_weights 分支，mgoin 回应分支逻辑必须保留在 kernel 内。HDCharles 质疑 as_quantized_activation API 的复杂度，mgoin 表示是风格选择可后续修改。zyongye 提议将 padding/alignment 编码进 QuantKey 以减少额外

kernel launch，mgoin 认为是后续课题。HDCharles 询问是否假设无 zero point，mgoin 确认现阶段无 fused kernel 使用 zero point。

- apply_mm 静态方法提议 (design): mgoin 认为分支逻辑必须保留在 kernel 内部，因为 kernel 知道自己的预处理（如 NVFP4 的 padding），提取成统一方法会丢失 kernel 特定信息。
- as_quantized_activation API 设计 (design): mgoin 认为是风格选择，as_quantized_activation 可以封装验证逻辑和拆包，隐藏细节。可后续修改。
- padding/alignment 加入 QuantKey (design): mgoin 认为是后续课题，且涉及融合 kernel 选型权衡（不同 fused kernel 可能支持不同 padding）。暂不实现。
- zero point 支持 (question): mgoin 确认目前 fused kernel 都不使用 zero point（仅 FP8、NVFP4），所以无需在 QuantKey 中表示 zero point。

风险与影响

- 风险：新契约为已知 kernel 设计了默认返回 None，不影响原有行为。但存在以下风险：1) 如果 kernel 声明了 input_quant_key 但 apply_weights 未调用 as_quantized_activation，输入会被当作普通张量处理导致静默错误。测试 test_supporting_backend_declares_consume_via_helper 通过字节码检查调用，但可能因装饰器或间接调用产生漏报。 2) 新接口增加了 kernel 实现者的认知负担，需要理解契约。 3) 未处理 layout/padding 编码，可能导致融合 producer 输出不符合 kernel 预期。
  - 影响：用户：无直接可见变化，但为后续手动融合优化建立了基础。系统：新增量化融合架构扩展点，线性 kernel 需声明 input_quant_key 并处理 QuantizedActivation。团队：需遵循此契约编写 fusion producer 代码；现有 kernel 若支持预量化需添加 input_quant_key 覆盖和 apply_weights 适配。
  - 风险标记：kernel 可能遗漏调用 as_quantized_activation，测试依赖字节码检测不稳健，layout/padding 未编码可能致错

关联脉络

- PR #43224 [RFC]: Porting compiler fusions to manual fusion: 本 PR 是为该 RFC 的第一部分，建立消费者侧契约，使后续手动融合能够避免线性层重复量化。