PR #41652 完整报告

vllm-project/vllm

[Quantization] add humming moe backend to all dense/moe oracles

合并时间：2026-07-07 04:36

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/41652

执行摘要

- 一句话：为所有 Dense/MoE 量化 oracle 添加 Humming 后端
- 推荐动作：此 PR 值得精读，特别是其 oracle 模式和权重转换架构。对于量化后端开发者和性能优化者，了解 Humming 后端的集成方式有助于未来添加更多后端。建议关注后续关于 Humming 性能基准的 PR 以及可能的默认启用决策。

功能与动机

量化模型需要高性能的内核后端来提升推理速度。Humming 是一个新的 GEMM 内核库，支持多种量化格式。此 PR 旨在将 Humming 后端集成到 vLLM 的量化框架中，使其能够被现有的 Dense 和 MoE oracle 选择，为用户提供更多后端选择并可能带来性能提升。PR body 中测试了包括 compressed-tensors、modelopt、fp8、gptq、awq 等在内的多种量化模型，覆盖 Dense 和 MoE 场景。

实现拆解

1. 新增 Humming 线性内核类：在 vllm/model_executor/kernels/linear/ 下为 FP8、INT8、NVFP4、MXFP4、MXFP8 各量化格式添加对应的 Humming 内核类（如 HummingFP8ScaledMMLinearKernel），继承自基类，实现 is_supported、can_implement、process_weights_after_loading 和 apply_weights 方法。这些类负责将标准权重转换为 Humming 内核期望的格式（如处理 scale 分组、全局尺度求逆），并调用 HummingMethod.forward_layer 执行实际计算。
2. 扩展 MoE oracle：在 vllm/model_executor/layers/fused_moe/oracle/ 下的各个 oracle 文件（fp8.py、int8.py、int_wna16.py、nvfp4.py 等）中添加对 Humming 后端的支持。具体包括：在后端枚举中加入 HUMMING 值；在 backend_to_kernel_cls 中返回 BatchedHummingGroupedExperts、HummingGroupedExperts、HummingIndexedExperts；实现 map_*_backend 函数以支持用户显式指定；在 make_*_moe_quant_config 中添加针对 Humming 的分支，调用 get_humming_moe_quant_config 生成量化配置。
3. 实现权重转换逻辑：在 oracle 中新增 _humming_*_weight_schema 函数，将标准的量化权重转换为 Humming 的 compressed-tensors schema 格式。例如，对于 INT8，将权重视为有符号 int8 并偏移 128，然后构建符合 compressed-tensors 格式的配置字典；对于 NVFP4，处理 weight_packed 重命名和全局尺度反转。

4. 调整量化方法：修改 auto_gptq.py、auto_awq.py、compressed_tensors_moe_wna16_marlin.py 等文件，在权重加载过程中当选择 Humming 后端时，跳过原有的内核特定加载，转而调用 Humming 的转换函数（如 convert_to_wna16_moe_kernel_format）。
5. 测试与依赖更新：添加了 lm-eval 测试（tests/evals/ 目录），更新 humming-kernels 版本依赖至 0.1.10，并在 CI 配置中添加了可选依赖标记。

关键文件：

- vllm/model_executor/kernels/linear/scaled_mm/humming.py（模块 线性内核；类别 source；类型 data-contract；符号 HummingFP8ScaledMMLinearKernel, is_supported, can_implement, process_weights_after_loading）：核心线性内核实现，定义了 FP8/INT8 的 Humming 内核类，包含权重转换和推理前向逻辑。
- vllm/model_executor/kernels/linear/nvfp4/humming.py（模块 线性内核；类别 source；类型 data-contract；符号 HummingNvFp4LinearKernel, is_supported, can_implement, process_weights_after_loading）：展示了 NVFP4 特定处理：全局尺度反转和 weight_packed 重命名。
- vllm/model_executor/layers/fused_moe/oracle/int_wna16.py（模块 MoE oracle；类别 source；类型 data-contract；符号 map_wna16_backend, _humming_wna16_weight_schema）：MoE oracle 的典型修改：添加 HUMMING 后端枚举、映射函数和 kernel class 返回。

关键符号：HummingFP8ScaledMMLinearKernel.is_supported, HummingFP8ScaledMMLinearKernel.process_weights_after_loading, HummingFP8ScaledMMLinearKernel.apply_weights, HummingInt8ScaledMMLinearKernel.process_weights_after_loading, HummingNvFp4LinearKernel.process_weights_after_loading, HummingMxFp4LinearKernel.process_weights_after_loading, HummingMxfp8LinearKernel.process_weights_after_loading, map_wna16_backend, _humming_wna16_weight_schema, _humming_fp8_weight_schema, _humming_int8_weight_schema

关键源码片段

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

核心线性内核实现，定义了 FP8/INT8 的 Humming 内核类，包含权重转换和推理前向逻辑。

```
# SPDX-License-Identifier: Apache-2.0
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project

import torch
from vllm.logger import init_logger
from vllm.model_executor.layers.quantization.utils.humming_utils import (
    convert_linear_layer_to_humming_standard,
    prepare_humming_layer,
)
```

```
from vllm.platforms import current_platform
from .ScaledMMLinearKernel import (
    FP8ScaledMMLinearKernel,
    FP8ScaledMMLinearLayerConfig,
    Int8ScaledMMLinearKernel,
    Int8ScaledMMLinearLayerConfig,
)

logger = init_logger(__name__)


class HummingFP8ScaledMMLinearKernel(FP8ScaledMMLinearKernel):
    """Humming GEMM 内核 for FP8 量化."""

    @classmethod
    def is_supported(cls, compute_capability: int | None = None) -> bool:
        # 仅支持 CUDA 且 SM >= 75
        if not current_platform.is_cuda():
            return False
        if not current_platform.has_device_capability(75):
            return False
        return True

    @classmethod
    def can_implement(cls, config: FP8ScaledMMLinearLayerConfig) -> bool:
        # 目前接受所有 FP8 配置
        return True

    def process_weights_after_loading(self, layer: torch.nn.Module) -> None:
        # 将标准权重转换为 Humming 格式
        from vllm.utils.humming import dtypes
        name_map = {"weight": "weight", "weight_scale": "weight_scale"}
        # 获取 scale 的数据类型，并映射到 Humming 的 dtype
        scale_torch_dtype = self.config.weight_quant_key.scale.dtype
        scale_dtype = dtypes.DataType.from_torch_dtype(scale_torch_dtype)
        # 构造量化配置字典
        quant_config = {
            "quant_method": "humming",
            "dtype": "float8e4m3",
            "scale_dtype": scale_dtype,
        }
        # 根据 scale 的 group_shape 决定 weight_scale_type
        scale_group_shape = self.config.weight_quant_key.scale.group_shape
        if scale_group_shape.is_per_tensor():
            quant_config["weight_scale_type"] = "tensor"
            # 如果 layer 没有 global_scale 但有 weight_scale，则重命名
            if not hasattr(layer, "global_scale") and hasattr(layer, "weight_scale"):
                del name_map["weight_scale"]
                name_map["global_scale"] = "weight_scale"
```

```
        elif scale_group_shape.is_per_channel():
            quant_config["weight_scale_type"] = "channel"
        elif scale_group_shape.is_per_group():
            quant_config["weight_scale_type"] = "group"
            quant_config["group_size"] = scale_group_shape.col
        else:
            # block 格式
            quant_config["weight_scale_type"] = "block"
            quant_config["weight_scale_group_size_n"] = scale_group_shape.row
            quant_config["weight_scale_group_size"] = scale_group_shape.col
            if hasattr(layer, "weight_scale_inv"):
                name_map["weight_scale"] = "weight_scale_inv"
        # 执行转换
        convert_linear_layer_to_humming_standard(layer=layer, name_map=name_map)
        prepare_humming_layer(layer, quant_config)

    def apply_weights(self, layer: torch.nn.Module, x: torch.Tensor,
                      bias: torch.Tensor | None = None) -> torch.Tensor:
        # 使用 HummingMethod 进行前向计算
        from vllm.utils.humming import HummingMethod
        flatten_inputs = x.view(-1, x.size(-1))
        output = HummingMethod.forward_layer(
            layer=layer,
            inputs=flatten_inputs,
            compute_config=layer.compute_config,
        )
        return output.view(*x.shape[:-1], output.size(-1))
```

vllm/model_executor/kernels/linear/nvfp4/humming.py

展示了 NVFP4 特定处理：全局尺度反转和 weight_packed 重命名。

```
# SPDX-License-Identifier: Apache-2.0
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project

import torch
from vllm.logger import init_logger
from vllm.model_executor.layers.quantization.utils.humming_utils import prepare_humming_
layer
from vllm.platforms import current_platform
from .base import NvFp4LinearKernel, NvFp4LinearLayerConfig

logger = init_logger(__name__)


class HummingNvFp4LinearKernel(NvFp4LinearKernel):
    """Humming GEMM 内核 for NVFP4 量化."""

    @classmethod
    def is_supported(cls, compute_capability: int | None = None) -> bool:
```

```
        if not current_platform.is_cuda():
            return False
        if not current_platform.has_device_capability(75):
            return False
        return True

    @classmethod
    def can_implement(cls, config: NvFp4LinearLayerConfig) -> bool:
        return True

    def process_weights_after_loading(self, layer: torch.nn.Module) -> None:
        # NVFP4 特殊处理：compressed-tensors 的 nvfp4 加载器期望 weight_packed 属性
        # 但模型可能只有 weight，需要重命名
        if not hasattr(layer, "weight_packed"):
            layer.weight_packed = layer.weight
            del layer.weight
        # compressed-tensors 的线性方案会对全局尺度求逆（1/scale），而 Humming 需要原始值
        # 所以这里还原：取倒数
        layer.weight_global_scale = torch.nn.Parameter(
            1.0 / layer.weight_global_scale, requires_grad=False
        )
        # 最后调用 prepare_humming_layer 初始化 Humming 层
        quant_config = {
            "quant_method": "compressed-tensors",
            "format": "nvfp4-pack-quantized",
            "type": "float",
            "num_bits": 4,
            "strategy": "group",
            "group_size": 16,
        }
        prepare_humming_layer(layer, quant_config)

    def apply_weights(self, layer: torch.nn.Module, x: torch.Tensor,
                      bias: torch.Tensor | None = None) -> torch.Tensor:
        from vllm.utils.humming import HummingMethod
        flatten_inputs = x.view(-1, x.size(-1))
        output = HummingMethod.forward_layer(
            layer=layer, inputs=flatten_inputs, compute_config=layer.compute_config
        )
        return output.view(*x.shape[:-1], output.size(-1))
```

vllm/model_executor/layers/fused_moe/oracle/int_wna16.py

MoE oracle 的典型修改：添加 HUMMING 后端枚举、映射函数和 kernel class 返回。

```
# 摘自 vllm/model_executor/layers/fused_moe/oracle/int_wna16.py

from vllm.config.kernel import MoEBackend
from vllm.model_executor.layers.fused_moe.modular_kernel import FusedMoEExperts

class WNA16MoEBackend(Enum):
```

```
    MARLIN = "MARLIN"
    BATCHED_MARLIN = "BATCHED_MARLIN"
    HUMMING = "HUMMING" # 新增 Humming 后端
    CPU = "CPU"
    FLASHINFER_TRTLLM = "FLASHINFER_TRTLLM"
    XPU = "XPU"

def backend_to_kernel_cls(backend: WNA16MoEBackend) -> list[type[FusedMoEExperts]]:
    """返回给定后端对应的 experts 类列表。"""
    if backend == WNA16MoEBackend.HUMMING:
        from vllm.model_executor.layers.fused_moe.experts.fused_humming_moe import (
            BatchedHummingGroupedExperts,
            HummingGroupedExperts,
            HummingIndexedExperts,
        )
        return [BatchedHummingGroupedExperts, HummingGroupedExperts,
        HummingIndexedExperts]
    # ... 其他已有的 backends

def map_wna16_backend(runner_backend: MoEBackend) -> WNA16MoEBackend:
    """将用户指定的 MoEBackend 映射到 WNA16MoEBackend。"""
    mapping = {
        "marlin": WNA16MoEBackend.MARLIN,
        "humming": WNA16MoEBackend.HUMMING,
        "flashinfer_trtllm": WNA16MoEBackend.FLASHINFER_TRTLLM,
    }
    if backend := mapping.get(runner_backend):
        return backend
    raise ValueError(f"moe_backend='{runner_backend}' is not supported for WNA16 MoE. "
                     f"Expected one of {list(mapping.keys())}.")
```

评论区精华

PR 的 review 评论全部由机器人自动生成，无实质人工讨论。gemini-code-assist[bot] 的评论总结了变更要点："Key changes include updating the backend oracles to support Humming, implementing weight conversion logic for Humming kernels, and refactoring quantization methods to pass the layer instance to kernel and configuration factories." 整体上，变更设计遵循 oracle 模式，将后端选择与量化方法解耦，权重转换逻辑集中在 oracle 中，量化方法保持通用。

- gemini-code-assist[bot] 的变更总结 (other): 无争议，机器人自动评论。

风险与影响

- 风险：Humming 内核默认禁用，仅当用户通过 --moe-backend humming 或 --linear-backend humming 显式启用时才会生效，因此对现有用户无直接影响。主要风险包括： 1) 硬件兼容性：Humming 仅支持 CUDA 且 SM ≥ 75，在较旧 GPU 上会自动 fallback，但需确保 fallback 路径正确。 2) 数值精度：权重转换逻辑（如 INT8 的 +128

偏移、NVFP4 的全局尺度求逆）可能引入数值偏差，需依赖充分测试。 3) 性能不确定性：虽然 Humming 旨在提升性能，但 PR 未提供基准测试数据，实际效果需验证。 4) 维护成本：新增 73 个文件，代码库复杂度增加，需要持续维护 Humming 后端的兼容性和性能。

- 影响：对用户：提供新的高性能内核选择，但需要手动启用；现有量化模型无需修改即可使用（fallback 到原有后端）。对系统：增加了对 humming-kernels 库的依赖，但默认不安装（标记为可选）。对团队：需要维护与 Humming 内核版本的同步，以及与其他后端的兼容性。影响范围中等，不改变默认推理路径。
- 风险标记：新后端默认禁用，仅 CUDA/SM75+，数值精度需验证，增加维护负担

关联脉络

- PR #43328 Enable B12x backend for non-gated MoEs (like Nemotron): Both PRs add new custom MoE backends (B12x vs Humming) following similar oracle integration pattern.
- PR #43994 [Kernel][Helion][1/N] Add Helion kernel for silu_and_mul_per_block_quant: Similar in adding a new performance-oriented kernel backend with dedicated kernel files and oracle integration.