PR #41394 完整报告

vllm-project/vllm

[Kernel][ROCm] Native W4A16 kernel for AMD RDNA3 (gfx1100) — fp16 + bf16

合并时间：2026-05-29 19:04

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/41394

执行摘要

- 一句话：RDNA3 原生 W4A16 内核，fp16/bf16 性能飞跃
- 推荐动作：值得所有 AMD ROCm 平台开发者精读，尤其关注 C++ HIP 内核实现（标量 dot-product vs WMMA 调度、LDS 双缓冲、LLVM 编译器 bug 变通）和 Python 监听门控模式。对于量化推理优化者，该 PR 提供了一个针对消费级 GPU 极致优化的参考案例。

功能与动机

在 RDNA3 上，唯一快速的 W4A16 路径是 ExLlama（仅 fp16），将 bf16 训练模型强制转为 fp16 会导致数值不稳定（inf/NaN），而 Triton W4A16 bf16 速度仅为 ExLlama fp16 的 1/3。本 PR 消除了这种折衷：提供原生 bf16 W4A16 路径，速度 205-345 tk/s（Triton bf16 的 2.5-4.2 倍），同时保留 fp16 路径并超越 ExLlama。

实现拆解

1. 平台门控与架构检测：在 vllm/platforms/rocm.py 中新增 on_gfx11、on_gfx1100、on_gfx1151 函数及模块级布尔标志（如 _ON_GFX11），通过 amdsmi 解析 GCN 架构字符串，避免 CUDA 初始化。内核的 can_implement() 利用 on_gfx1100() 精确限制在目标硬件。
2. Python 内核封装类：新增 vllm/model_executor/kernels/linear/mixed_precision/rdna3_w4a16.py，实现 RDNA3W4A16LinearKernel 继承自 MPLinearKernel。包含 process_weights_after_loading（处理权重重排、零点点合成）、apply_weights（调用 HIP 操作）等方法，并注册到内核优先级列表（优先于 TritonW4A16LinearKernel）。
3. C++ HIP 内核实现：在 csrc/rocm/ 下添加两个 CU 文件：
   - q_gemm_rdna3.cu：标量 dot-product 内核，用于 decode（M<16）和 fp16 路径，使用 v_dot2_f32_f16 和 v_dot2_f32_bf16（通过 union 指针绕过 LLVM 折叠 bug）。
   - q_gemm_rdna3_wmma.cu：WMMA 矩阵乘法内核，用于 prefill（M≥16），支持 v_wmma_f32_16x16x16 指令，包含多个 tile 版本（16x16_1w 至 128x64_k32），通过 K 分割和 LDS 双缓冲优化。 共享反量化工具头文件 qdq_4_rdna3.cuh。操作通过 TORCH_LIBRARY_EXPAND 注册到 _rocm_C 扩展。
4. Python 操作绑定与编译集成：在 vllm/_custom_ops.py 中定义 gptq_gemm_rdna3 函数和两个 fake 操作（用于 torch.compile），并在 csrc/rocm/torch_bindings.cpp 和 ops.h 中注册。内核文件通过 CMake 构建系统编译。

5. 测试与验证：新增两个测试文件：
   - test_rdna3_w4a16.py：端到端正确性测试，对比 fp32 参考（dequant+matmul）验证 kernel 输出，覆盖 fp16/bf16、有无零点、多种形状（M=1,16,128 等）。
   - test_rdna3_w4a16_selection.py：内核选择测试，验证在 gfx1100 上 choose_mp_linear_kernel 优先返回 RDNA3 内核，以及 can_implement 正确拒绝不支持的配置。

关键文件：

- vllm/model_executor/kernels/linear/mixed_precision/rdna3_w4a16.py（模块 量化内核层；类别 source；类型 core-logic；符号 RDNA3W4A16LinearKernel, get_min_capability, can_implement, process_weights_after_loading）：核心 Python 封装类，定义 RDNA3W4A16LinearKernel，包含权重预处理、apply_weights 以及内核门控逻辑（can_implement）。是内核与 vLLM 推理框架的接口。
- tests/kernels/quantization/test_rdna3_w4a16.py（模块 测试；类别 test；类型 test-coverage；符号 _reference, _build_layer, DummyLayer, _run_kernel）：主要正确性测试文件，覆盖 fp16/bf16、有无零点、多种 M/K/N 形状，通过 fp32 参考对比验证内核输出。
- vllm/_custom_ops.py（模块 操作绑定；类别 source；类型 core-logic；符号 gptq_gemm_rdna3, _gptq_gemm_rdna3_fake, _gptq_gemm_rdna3_wmma_fake）：注册新的 ROCm 操作 gptq_gemm_rdna3 及其 fake 实现，是 Python 与 C++ 内核的桥梁。

关键符号：RDNA3W4A16LinearKernel.can_implement, RDNA3W4A16LinearKernel.process_weights_after_loading, RDNA3W4A16LinearKernel.apply_weights, vllm._custom_ops.gptq_gemm_rdna3, vllm.platforms.rocm.on_gfx1100, vllm.platforms.rocm.on_gfx11, vllm.platforms.rocm.on_gfx1151, tests.kernels.quantization.test_rdna3_w4a16._reference, tests.kernels.quantization.test_rdna3_w4a16.test_rdna3_w4a16_matches_reference, tests.kernels.quantization.test_rdna3_w4a16.test_rdna3_w4a16_bias, tests.kernels.quantization.test_rdna3_w4a16_selection.test_selection_prefers_rdna3, tests.kernels.quantization.test_rdna3_w4a16_selection.test_can_implement

关键源码片段

vllm/model_executor/kernels/linear/mixed_precision/rdna3_w4a16.py

核心 Python 封装类，定义 RDNA3W4A16LinearKernel，包含权重预处理、apply_weights 以及内核门控逻辑（can_implement）。是内核与 vLLM 推理框架的接口。

```
# SPDX-License-Identifier: Apache-2.0
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project
"""W4A16 GPTQ kernel for AMD RDNA3 (gfx1100) — fp16 + bf16."""

import torch
from vllm import _custom_ops as ops
from vllm.model_executor.layers.quantization.utils.quant_utils import pack_quantized_values_
into_int32
```

```
from vllm.model_executor.parameter import BasevLLMParameter, permute_param_layout_
from vllm.platforms import current_platform
from vllm.scalar_type import scalar_types
from .MPLinearKernel import MPLinearKernel, MPLinearLayerConfig


class RDNA3W4A16LinearKernel(MPLinearKernel):
    SUPPORTED_QUANT_TYPES = [scalar_types.uint4b8]

    @classmethod
    def get_min_capability(cls) -> int:
        return 60 # ROCm 门控通过 can_implement 内的 on_gfx1100() 实现

    @classmethod
    def can_implement(cls, c: MPLinearLayerConfig) -> tuple[bool, str | None]:
        # 仅在 ROCm 且 gfx1100 上激活
        if not current_platform.is_rocm():
            return False, "RDNA3 W4A16 kernel is ROCm-only"
        from vllm.platforms.rocm import on_gfx1100
        if not on_gfx1100():
            return False, "RDNA3 W4A16 kernel requires gfx1100"
        # 检查自定义操作是否已编译
        if not (hasattr(torch.ops, "_rocm_C") and hasattr(torch.ops._rocm_C, "gptq_gemm_rdna3"))
        :
            return False, "torch.ops._rocm_C.gptq_gemm_rdna3 missing — rebuild C++ extension"
        if c.act_type not in (torch.float16, torch.bfloat16):
            return False, "RDNA3 W4A16 kernel only supports fp16 and bf16"
        if c.weight_type not in cls.SUPPORTED_QUANT_TYPES:
            return False, f"Quant type ({c.weight_type}) not supported"
        if c.group_size <= 0:
            return False, "does not support channelwise quantization"
        if c.full_weight_shape[0] % c.group_size != 0:
            return False, "Group size does not divide K"
        # N 必须是 8 的倍数，满足 qzeros 打包对齐
        if c.partition_weight_shape[1] % 8 != 0:
            return False, "Output features must be a multiple of 8"
        if c.has_g_idx and c.partition_weight_shape[0] != c.full_weight_shape[0]:
            return False, "Act-order with TP-partitioned not supported"
        return True, None

    def process_weights_after_loading(self, layer: torch.nn.Module):
        """权重预处理：合成零点（如果不存在）、打乱布局，与 ExLlama 布局兼容。"""
        c = self.config
        device = getattr(layer, self.w_q_name).device
        if not c.zero_points:
            self.w_zp_name = "qzeros"
            groups = c.partition_weight_shape[0] // c.group_size
            out_features = c.partition_weight_shape[1]
            if c.weight_type.has_bias():
```

```
                # GPTQv1 约定：存储零点 = bias - 1，内核运行时加 1
                zeros = torch.full((groups, out_features), c.weight_type.bias - 1,
                                   dtype=torch.int32, device=device)
            else:
                raise NotImplementedError("zero-bias 4-bit quant requires explicit zero points")
            zeros = pack_quantized_values_into_int32(zeros, c.weight_type, packed_dim=1)
            setattr(layer, self.w_zp_name, zeros)
        # 调用基类处理权重重排
        super().process_weights_after_loading(layer)

    def apply_weights(self, layer: torch.nn.Module, x: torch.Tensor, bias: torch.Tensor | None =
    None) -> torch.Tensor:
        """执行 W4A16 矩阵乘法。"""
        w_q = getattr(layer, self.w_q_name)
        w_s = getattr(layer, self.w_s_name)
        w_zp = getattr(layer, self.w_zp_name, None) if self.config.zero_points else None
        g_idx = getattr(layer, self.w_g_idx_name, None)
        return ops.gptq_gemm_rdna3(x, w_q, w_zp, w_s, g_idx, use_v2_format=self.config.zero_
        points)
```

tests/kernels/quantization/test_rdna3_w4a16.py

主要正确性测试文件，覆盖 fp16/bf16、有无零点、多种 M/K/N 形状，通过 fp32 参考对比验证内核输出。

```
# 核心参考实现，用于对比 kernel 输出
# 正确性测试的核心：先手动 dequant，然后执行 fp32 matmul

def _reference(
    x_mk: torch.Tensor,
    q_int4_kn: torch.Tensor,
    scales_gn: torch.Tensor,
    zeros_gn: torch.Tensor | None, # None 表示对称路径，kernel 合成零点 = 7
    group_size: int,
    bias: torch.Tensor | None,
) -> torch.Tensor:
    K, N = q_int4_kn.shape
    s_full = scales_gn.repeat_interleave(group_size, dim=0).to(torch.float32) # [K, N]
    if zeros_gn is None:
        # kernel 内部：存储零点 = 7，实际零点 = 7 + 1 = 8（weight_type.bias）
        z_full = torch.full((K, N), float(WEIGHT_TYPE.bias), device=x_mk.device, dtype=torch.
        float32)
    else:
        # 应用 GPTQv1 +1 约定
        z_full = (zeros_gn + 1).repeat_interleave(group_size, dim=0).to(torch.float32)
    w_fp = (q_int4_kn.to(torch.float32) - z_full) * s_full # [K, N] 反量化
    out = x_mk.to(torch.float32) @ w_fp # [M, N]
    if bias is not None:
        out = out + bias.to(torch.float32)
    return out.to(x_mk.dtype)
```

```
@gfx1100_only
@pytest.mark.parametrize("dtype", [torch.float16, torch.bfloat16])
@pytest.mark.parametrize("has_zp", [True, False])
@pytest.mark.parametrize("M,K,N,group_size", [
    (1, 4096, 256, 128),
    (16, 4096, 256, 128),
    (128,4096, 256, 128),
    (1, 5120, 128, 128),
    (1, 11008,128, 128),
])
def test_rdna3_w4a16_matches_reference(M, K, N, group_size, has_zp, dtype):
    """端到端测试：构建 GPTQ 层 -> 预处理权重 -> kernel 推理 -> 与参考对比"""
    layer, ctx = _build_layer(M, K, N, group_size, has_zp, dtype)
    kernel = RDNA3W4A16LinearKernel(ctx)
    kernel.process_weights_after_loading(layer)
    x = torch.randn(M, K, dtype=dtype, device=device)
    out = kernel.apply_weights(layer, x)
    ref = _reference(x, ctx["q_int4_kn"], ctx["scales_gn"], ctx["zeros_gn"], group_size, bias=ctx.
    get("bias"))
    # 允许 fp16/bf16 精度差异，rtol 设为 1e-2, atol 1e-2
    assert torch.allclose(out, ref, rtol=1e-2, atol=1e-2), f"Mismatch for M={M}, K={K}, N={N}"
```

vllm/_custom_ops.py

注册新的 ROCm 操作 gptq_gemm_rdna3 及其 fake 实现，是 Python 与 C++ 内核的桥梁。

```
# 在 vllm/_custom_ops.py 中（原有 gptq_gemm 函数之后）

def gptq_gemm_rdna3(
    a: torch.Tensor,
    b_q_weight: torch.Tensor,
    b_qzeros: torch.Tensor,
    b_scales: torch.Tensor,
    b_g_idx: torch.Tensor,
    use_v2_format: bool,
) -> torch.Tensor:
    """封装 _rocm_C 操作，支持 torch.compile 动态 shape。"""
    return torch.ops._rocm_C.gptq_gemm_rdna3(
        a, b_q_weight, b_qzeros, b_scales, b_g_idx, use_v2_format
    )

# 如果操作存在，注册 fake 实现（用于 torch.compile shape 推导）
if hasattr(torch.ops, "_rocm_C") and hasattr(torch.ops._rocm_C, "gptq_gemm_rdna3"):

    @register_fake("_rocm_C::gptq_gemm_rdna3")
    def _gptq_gemm_rdna3_fake(
        a: torch.Tensor,
        b_q_weight: torch.Tensor,
        b_qzeros: torch.Tensor,
        b_scales: torch.Tensor,
```

```
        b_g_idx: torch.Tensor,
        use_v2_format: bool,
    ) -> torch.Tensor:
        return torch.empty(
            (a.size(0), b_q_weight.size(1)), dtype=a.dtype, device=a.device
        )
```

评论区精华

- 维护担忧（AndreasKaratzas）：CI 没有 RDNA3 设备，回归可能无法检测，是否应该上游。JartX 回应代码隔离在专属模块且不影响其他路径；最终 tjtanaa 批准。
- gfx1151 兼容性（gshtras）：gfx1151（Strix Halo）是否应被包含？JartX 随后将内核严格门控到 gfx1100（commit 35d575a），并添加 on_gfx1151() 辅助函数供混合内核使用。
- act-order 排列缺失（depthfirst-app）：V7/V8 WMMA 内核忽略 b_q_perm，导致 act-order 模型在 prefill 阶段输出错误。JartX 在 commit 649688c 中修复，将 act-order 请求路由到 V5（该 tile 处理排列）。
- 文件位置与编译门控（mgehre-amd）：建议将内核从 csrc/libtorch_stable/ 移到 csrc/quantization/gptq/ 或 _rocm_C 扩展以解决 NVCC 兼容性。JartX 随后重构（commit 7e66513）将文件移至 csrc/rocm/，并在构建系统中正确隔离。
- 用户反馈（wizardeur, gesong2077）：在实际编码工作负载中验证了 PR 的稳定性和性能提升，社区期望合并。
- 维护责任与 CI 覆盖 (design): 代码隔离在专属模块，且仅 gfx1100 激活；团队接受维护风险。
- gfx1151 (Strix Halo) 兼容性 (design): 内核仅限 gfx1100；gfx1151 留给混合内核（PR #40977）。
- WMMA 内核忽略 act-order 排列 (correctness): JartX 提交修复（commit 649688c）：将 act-order 请求路由到 V5 tile（该 tile 正确应用排列）。
- 文件位置与编译门控 (design): 文件移至 csrc/rocm/ 并正确门控，确保 NVCC 下安全存根。
- 标量内核 bf16 M=1 跳过 act-order (correctness): JartX 在后续 commits 中修复：为 bf16 M=1 路径恢复 LDS staging（或通过其他方式应用排列）。最终提交 05f48afa 明确应用 act-order 排列。
- 使用 constexpr 替换宏 (style): 未被采纳（可能因 HIP 内核风格或兼容性原因），评论已解决。

风险与影响

- 风险：
  - 回归风险：内核仅在 gfx1100 上激活，但 CI 缺少 RDNA3 设备，任何回归（如编译失败、性能下降）在合并前可能无法被自动检测。
  - 正确性风险：早期版本存在 act-order 排列未应用和 bf16 M=1 路径直接读取全局内存的问题，虽已在最终提交中修复，但若未来代码重构导致类似逻辑退化，可能重新引入。此外，C++ 操作缺少对 N%8 的 TORCH_CHECK（依赖 Python 层），存在潜在的内存越界。

- 编译风险：内核使用 ROCm 内置函数（如 __builtin_amdgcn_fdot2、WMMA 内置），需确保在非 ROCm 编译环境下安全存根。已通过 _rocm_C 扩展和 #if defined(USE_ROCM) 门控。
- 性能回退：多版本 WMMA 内核（V1-V8）的分派逻辑复杂，可能在某些边角情况（如 K 余数处理）触发次优路径或原子 CAS 竞争。测试已覆盖常见形状，但不排除发布后出现性能退化。
- 影响：对 AMD RDNA3 (gfx1100) 用户有巨大正面影响：W4A16 推理延迟降低 2.5-4.2 倍，支持 bf16 保留模型原生精度；fp16 路径同样超越 ExLlama。对其他 AMD GPU（CDNA、RDNA4）无影响——自动回退到 Triton 内核。团队需承担维护责任，但代码隔离在 rdna3_w4a16.py 和 csrc/rocm/ 下，不干扰核心路径。新增 ~3800 行代码，其中 2200 行为 C++ 内核，测试覆盖充分。
- 风险标记：缺少 RDNA3 CI 覆盖，act-order 边角情形已修复，编译门控依赖特定架构，无回归自动检测机制

关联脉络

- PR #40977 [ROCm][W4A16] Hybrid kernel for gfx1151 (Strix Halo) — HIP decode + Triton prefill: mgehre-amd 提出的混合 W4A16 内核 PR，与本文 RDNA3 内核在相同功能域（W4A16 on ROCm），两者在 review 中进行了大量性能对比和设计讨论，并计划将本 PR 的 decode 路径集成到混合内核中。