PR #22685 完整报告

sgl-project/sglang

[CPU] [Quantization] Add GPTQ/AWQ 4bits quantization support for CPU

合并时间：2026-04-23 04:34

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/22685

执行摘要

- 一句话：为 CPU 平台添加 GPTQ/AWQ 4 位量化支持，扩展 Intel AMX 后端能力。
- 推荐动作：建议技术管理者和工程师精读此 PR，重点关注 CPU 量化配置与内核集成的设计决策，如 CPUQuantAlgo 枚举的使用和权重处理流程的分支逻辑，这些对于理解跨平台量化支持架构有较高参考价值。

功能与动机

PR body 指出目标是添加 CPU 平台的 AWQ/GPTQ 格式支持，以扩展量化功能到非 GPU 设备，特别是针对 Intel AMX 优化。作者提到此 PR 包含已关闭的 PR #8225，旨在解决 CPU 上 GPTQ/AWQ 权重的解包、重打包和内核集成问题。

实现拆解

1. 新增 CPU 专用配置类：在 python/sglang/srt/layers/quantization/ 下添加 gptq_cpu.py 和 awq_cpu.py，定义 CPUGPTQConfig 和 CPUAWQConfig 类，继承自基础配置，覆盖 get_supported_act_dtypes 和 get_quant_method 方法，以返回 CPU 专用的线性方法。
2. 实现线性与 MoE 方法：在新增文件中定义 GPTQLinearIntelAMXMethod 和 AWQLinearIntelAMXMethod 等类，处理权重的创建（create_weights）、后加载处理（process_weights_after_loading）和前向应用（apply），后者调用底层内核如 torch.ops.sgl_kernel.int4_scaled_mm_cpu。
3. 扩展 AMX 工具函数：修改 amx_utils.py，添加 CPUQuantAlgo 枚举以区分 AWQ 和 GPTQ，并更新 _amx_process_weight_after_loading 函数，支持通过 qweight_packed_method 参数处理量化权重的重打包逻辑。
4. 更新 C++ 内核：修改 sgl-kernel/csrc/cpu/gemm_int4.cpp，新增 unpack_4bit_to_32bit_signed 和 autogptq_to_int4pack 函数，实现 GPTQ 格式的解包，并调整 int4pack 函数以根据 CPUQuantAlgo 选择 AWQ 或 GPTQ 处理路径。
5. 注册与测试配套：更新 __init__.py 以注册 CPU 量化方法到 CPU_QUANTIZATION_METHODS，并修改测试文件如 test/srt/cpu/test_gemm.py，添加 test_int4_gptq_gemm 等测试用例，确保功能覆盖。

关键文件：

- python/sglang/srt/layers/quantization/gptq_cpu.py（模块 量化层；类别 source；类型 core-logic；符号 CPUGPTQConfig, GPTQLinearIntelAMXMethod,

get_supported_act_dtypes, get_quant_method）：新增 GPTQ 在 CPU 上的核心实现，包含配置类和线性方法，是功能的主要入口。

- python/sglang/srt/layers/quantization/awq_cpu.py（模块 量化层；类别 source；类型 core-logic；符号 CPUAWQConfig, AWQLinearIntelAMXMethod, is_layer_skipped_awq, get_supported_act_dtypes）：新增 AWQ 在 CPU 上的核心实现，包含配置类、线性方法和 MoE 方法。
- python/sglang/srt/layers/amx_utils.py（模块 工具函数；类别 source；类型 core-logic；符号 CPUQuantAlgo, _amx_process_weight_after_loading）：扩展 AMX 工具函数以支持量化算法区分，是权重重打包的关键逻辑。
- sgl-kernel/csrc/cpu/gemm_int4.cpp（模块 内核实现；类别 source；类型 core-logic；符号 unpack_4bit_to_32bit_signed, autogptq_to_int4pack, int4pack）：修改 C++ 内核以支持 GPTQ 解包和量化算法分发，是性能关键路径。
- python/sglang/srt/layers/quantization/__init__.py（模块 量化注册；类别 source；类型 dependency-wiring）：更新量化方法注册逻辑，添加 CPU 专用方法并修复拼写错误。

关键符号：CPUGPTQConfig.get_supported_act_dtypes, GPTQLinearIntelAMXMethod.apply, CPUAWQConfig.get_quant_method, AWQLinearIntelAMXMethod.process_weights_after_loading, _amx_process_weight_after_loading

关键源码片段

python/sglang/srt/layers/quantization/gptq_cpu.py

新增 GPTQ 在 CPU 上的核心实现，包含配置类和线性方法，是功能的主要入口。

```
from __future__ import annotations
from typing import TYPE_CHECKING, List, Optional
import torch
from sglang.srt.layers.quantization.base_config import LinearMethodBase
from .gptq import GPTQConfig

class CPUGPTQConfig(GPTQConfig):
    """CPU 专用的 GPTQ 配置类，继承自基础 GPTQConfig。"""

    @classmethod
    def get_supported_act_dtypes(cls) -> List[torch.dtype]:
        # 支持半精度和 bfloat16 激活数据类型，适用于 CPU AMX 后端
        return [torch.half, torch.bfloat16]

    def get_quant_method(
        self, layer: torch.nn.Module, prefix: str
    ) -> Optional[LinearMethodBase]:
        # 延迟导入以避免循环依赖
        from sglang.srt.layers.linear import LinearBase
        from sglang.srt.layers.moe.fused_moe_triton import FusedMoE

        if isinstance(layer, FusedMoE):
```

```
            return GPTQMoEIntelAMXMethod(self) # 返回 MoE 方法
        if isinstance(layer, LinearBase):
            return GPTQLinearIntelAMXMethod(self) # 返回线性方法
        return None


class GPTQLinearIntelAMXMethod(LinearMethodBase):
    """用于 Intel CPU AMX 的 GPTQ 线性方法。"""

    def __init__(self, quant_config: GPTQConfig):
        self.quant_config = quant_config
        # GPTQ v1 和 v2 格式处理零点的方式不同
        self.use_v2_format = quant_config.checkpoint_format == "gptq_v2"

    def apply(
        self,
        layer: torch.nn.Module,
        x: torch.Tensor,
        bias: Optional[torch.Tensor] = None,
    ) -> torch.Tensor:
        # 调用底层 CPU 内核进行 4 位量化矩阵乘法
        return torch.ops.sgl_kernel.int4_scaled_mm_cpu(
            x,
            layer.qweight,
            layer.qzeros,
            layer.scales,
            bias,
        )
```

python/sglang/srt/layers/quantization/awq_cpu.py

新增 AWQ 在 CPU 上的核心实现，包含配置类、线性方法和 MoE 方法。

```
from __future__ import annotations
import logging
from typing import TYPE_CHECKING, List, Optional
import torch
from sglang.srt.layers.quantization.base_config import LinearMethodBase
from .awq import AWQConfig, AWQLinearMethod

logger = logging.getLogger(__name__)

def is_layer_skipped_awq(prefix: str, modules_to_not_convert: List[str]):
    # 检查前缀是否在跳过转换的模块列表中，用于过滤特定层
    return any(module_name in prefix for module_name in modules_to_not_convert)


class CPUAWQConfig(AWQConfig):
    """CPU 专用的 AWQ 配置类，继承自基础 AWQConfig。"""

    def get_supported_act_dtypes(self) -> List[torch.dtype]:
```

```
        # 支持 float16 和 bfloat16 激活数据类型
        return [torch.float16, torch.bfloat16]

    def get_quant_method(
        self, layer: torch.nn.Module, prefix: str
    ) -> Optional[LinearMethodBase]:
        from sglang.srt.layers.linear import LinearBase
        from sglang.srt.layers.moe.fused_moe_triton import FusedMoE

        if isinstance(layer, LinearBase):
            if is_layer_skipped_awq(prefix, self.modules_to_not_convert):
                return UnquantizedLinearMethod() # 跳过量化
            return AWQLinearIntelAMXMethod(self) # 返回 AWQ 线性方法
        elif isinstance(layer, FusedMoE):
            return AWQMoEIntelAMXMethod(self) # 返回 AWQ MoE 方法
        return None
```

python/sglang/srt/layers/amx_utils.py

扩展 AMX 工具函数以支持量化算法区分，是权重重打包的关键逻辑。

```
from enum import IntEnum
import torch

class CPUQuantMethod(IntEnum):
    UNQUANT = 0
    INT8_W8A8 = 1
    FP8_W8A16 = 2
    INT4_W4A8 = 3

class CPUQuantAlgo(IntEnum):
    """CPU 量化算法枚举，用于区分 AWQ 和 GPTQ 格式。"""
    AWQ = 0
    GPTQ = 1

def _amx_process_weight_after_loading(
    module, weight_names, transpose_dims=None, qweight_packed_method=None
) -> None:
    # 根据量化方法选择处理路径
    if qweight_packed_method is None:
        # 原始非量化权重处理逻辑
        for i, weight_name in enumerate(weight_names):
            weight_tensor = getattr(module, weight_name)
            # ... 维度检查和重打包
    else:
        # 量化权重处理：调用内核函数进行重打包
        assert qweight_packed_method in ["awq", "gptq"]
        qweight_tensor = getattr(module, weight_names[0])
        qzeros_tensor = getattr(module, weight_names[1])
        scales_tensor = getattr(module, weight_names[2])
```

```
qweight, qzeros, scales = torch.ops.sgl_kernel.convert_weight_packed_scale_zp(
    qweight_tensor,
    qzeros_tensor,
    scales_tensor,
    CPUQuantAlgo.AWQ if qweight_packed_method == "awq" else CPUQuantAlgo.GPTQ,
)
# 将重打包后的权重设置为模块参数
packed_qweight = torch.nn.Parameter(qweight.detach(), requires_grad=False)
packed_qweight.__dict__ = qweight_tensor.__dict__
setattr(module, weight_names[0], packed_qweight)
# 类似处理 qzeros 和 scales
```

评论区精华

review 中 gemini-code-assist[bot] 指出高风险问题：在 gemm_int4.cpp 中硬编码的 + 1 偏移可能破坏 GPTQ v2 格式兼容性，作者回应已在 frontend 添加检查解决。Fridge003 建议为 CPU 创建独立文件以避免条件判断，作者已通过新增 awq_cpu.py 和 gptq_cpu.py 实现。此外，讨论还涉及拼写错误修正（如 CPU_QUANTIZATIPON_METHODS）、移除调试代码和冗余 .clone() 调用等优化建议。

- GPTQ v2 格式偏移问题 (correctness): 作者回应已在 frontend 添加检查来解决，但 review 中未显示具体修改，可能部分解决。
- 代码组织建议 (design): 作者已通过新增 awq_cpu.py 和 gptq_cpu.py 文件实现，减少了代码耦合。
- 拼写错误和代码清理 (style): 作者在后续提交中可能已修正，但 review 评论显示问题被识别。

风险与影响

- 风险：技术风险包括：GPTQ v2 格式的偏移处理可能未完全覆盖所有场景，导致模型加载错误；CPU AMX 支持依赖特定硬件（如 Intel 处理器），在不支持 AMX 的设备上可能回退到非优化路径，影响性能；张量并行填充逻辑在 create_weights 中的维度对齐检查可能引发运行时错误，特别是当 TP 大小不匹配时。
- 影响：对用户：使 CPU 用户能够部署 GPTQ/AWQ 4 位量化模型，降低内存占用并提升推理效率，扩展了 SGLang 的硬件支持范围。对系统：增加了 CPU 量化路径，可能引入新的代码复杂性和维护负担，但通过独立文件设计减少了与 GPU 代码的耦合。对团队：需要后续测试确保跨平台兼容性，并可能影响量化相关文档和示例更新。
- 风险标记：GPTQ v2 兼容性风险，硬件依赖风险，维度对齐风险

关联脉络

- PR #8225 未知（根据 PR body 提及）：此 PR 包含 PR #8225 的内容，且 #8225 已关闭，表明这是功能整合或重构。
- PR #23467 fix: dot-boundary match in is_layer_skipped for FP8 modules_to_not_convert: 同属量化模块修复，涉及模块路径匹配逻辑，可参考量化配置的通用模式。