PR #21954 完整报告

sgl-project/sglang

[1/4] NVFP4 KV cache: quantization strategy abstraction and kernel

合并时间：2026-04-29 16:45

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/21954

执行摘要

- 一句话：实现 NVFP4 KV cache 量化策略抽象与核心内核
- 推荐动作：值得精读，该 PR 展示了策略模式在推理引擎量化层的典型应用，接口设计清晰（抽象方法、属性、生命周期方法）。建议重点关注 dequantize_prev_kv 的返回值约定（FP8 dtype）以及 needs_dequant_workspace 标志位设计，同时留意 CUDA Graph 兼容性注释的演变以理解推理引擎对量化操作的特殊约束。阅读后可跟踪后续 PR 的完整数据流。

功能与动机

支持 SM120 GPU 上的 NVFP4 KV Cache 量化，通过 4 比特存储降低显存占用并提升解码吞吐（PR 基准测试显示 NVFP4 KV Cache 在解码延迟上比 FP8 提升 1.18 倍）。该 PR 将原 #21601 拆分为多部分进行增量评审，本部分聚焦量化策略抽象和内核工具。

实现拆解

1. 定义量化策略基类和注册机制：新增 fp4_kv_cache_quant_method.py，定义 FP4KVCacheQuantMethod 抽象基类，声明 create_buffers、quantize_and_store、dequantize_prev_kv、compute_cell_size 等核心接口。同时建立 FP4_KV_CACHE_QUANT_REGISTRY 字典和工厂函数 get_fp4_kv_cache_quant_method，将策略名称映射到实现类。
2. 实现 NVFP4 双层缩放策略：NVFP4KVMethod 实现全局 FP32 缩放（每层独立）和基于 FlashInfer nvfp4_kv_quantize / nvfp4_kv_dequantize 的块缩放（块大小 16）。needs_dequant_workspace 返回 True 以分配 FP8 反量化工作缓冲区（因目前尚无原生 FP4 prefill 内核）。
3. 实现 BlockFP4 单层缩放策略：BlockFP4KVMethod 实现类似 MXFP4 但块大小为 16 的单层缩放，使用纯 PyTorch 操作（batched_quantize / batched_dequantize），CPU 可测试。
4. 扩展量化工具类：在 kvfp4_tensor.py 中新增 NVFP4KVQuantizeUtil 封装 FlashInfer 内核的量化 / 反量化，支持 SM100+ 原生操作和 SM90 fallback。原有的 KVFP4QuantizeUtil 保留为 BlockFP4KVQuantizeUtil 的向后兼容别名。
5. 编写单元测试：新增 test_fp4_kv_cache_quant_method.py，包含注册表验证、工厂方法测试、NVFP4Method 和 BlockFP4Method 的缓冲区形状和精度往返测试，CI 注册为 CPU stage（CUDA 标记为 skip）。

关键文件：

- python/sglang/srt/layers/quantization/fp4_kv_cache_quant_method.py（模块 量化层；类别 source；类型 dependency-wiring；符号 FP4KVCacheQuantMethod, NVFP4KVMethod, BlockFP4KVMethod, FP4_KV_CACHE_QUANT_REGISTRY）：新增 FP4KVCacheQuantMethod 抽象基类和两个具体实现（NVFP4KVMethod、BlockFP4KVMethod），定义了量化缓存方法的完整接口和策略注册机制，是系列 PR 的核心架构基础。
- python/sglang/srt/layers/quantization/kvfp4_tensor.py（模块 量化内核；类别 source；类型 core-logic；符号 FP4KVCacheRecipe, BlockFP4KVQuantizeUtil, NVFP4KVQuantizeUtil, KVFP4QuantizeUtil）：新增 NVFP4KVQuantizeUtil（FlashInfer 内核集成）和 BlockFP4KVQuantizeUtil（纯 PyTorch 块级量化），同时引入 FP4KVCacheRecipe 枚举统一 FP4 格式标识，是量化工具的核心实现。
- test/registered/unit/layers/quantization/test_fp4_kv_cache_quant_method.py（模块 测试；类别 test；类型 test-coverage；符号 skip_if_no_cuda, TestKVCacheQuantRegistry, test_registry_contains_nvfp4_and_mxfp4, test_factory_nvfp4）：新增完整的单元测试，覆盖注册表查找、工厂方法、NVFP4Method 和 BlockFP4Method 的缓冲区形状、cell 大小、全局缩放初始化和 CUDA 下的量化反量化往返验证，确保新增模块的正确性。

关键符号：FP4KVCacheQuantMethod, NVFP4KVMethod, BlockFP4KVMethod, NVFP4KVQuantizeUtil.quantize, NVFP4KVQuantizeUtil.dequantize, BlockFP4KVQuantizeUtil.batched_quantize, BlockFP4KVQuantizeUtil.batched_dequantize

关键源码片段

python/sglang/srt/layers/quantization/fp4_kv_cache_quant_method.py

新增 FP4KVCacheQuantMethod 抽象基类和两个具体实现（NVFP4KVMethod、BlockFP4KVMethod），定义了量化缓存方法的完整接口和策略注册机制，是系列 PR 的核心架构基础。

```
# fp4_kv_cache_quant_method.py — 策略模式抽象基类与 NVFP4 实现

from abc import ABC, abstractmethod
from typing import Optional
import torch

class FP4KVCacheQuantMethod(ABC):
    """
    抽象基类：定义量化方法的标准接口。
    所有操作使用 FlashInfer 内核或纯张量操作，保持 CUDA Graph 兼容。
    """
    name: str
    SCALE_BLOCK_SIZE: int = 1 # 默认块大小

    def needs_dequant_workspace(self) -> bool:
        """是否需要分配反量化工作缓冲区（FP8 格式）用于 prefill。"""
```

```
        return False

    def needs_global_scale(self) -> bool:
        """是否使用每层全局 FP32 缩放。"""
        return False

    @abstractmethod
    def create_buffers(self, size: int, head_num: int, head_dim: int, layer_num: int, device: str) -
    > dict:
        """分配并返回缓冲区字典：
        k_buffer/v_buffer (FP4 打包), k_scale_buffer/v_scale_buffer, dq_k/dq_v 缓冲区。
        """
        ...

    @abstractmethod
    def quantize_and_store(self, k_buffer, v_buffer, k_scale_buffer, v_scale_buffer, loc, cache_k,
    cache_v, k_scale=None, v_scale=None) -> None:
        """量化 cache_k/cache_v 并写入缓冲区指定位置 loc。"""
        ...

    @abstractmethod
    def dequantize_prev_kv(self, k_fp4, k_scales, v_fp4, v_scales, layer_id) -> tuple[torch.Tensor,
    torch.Tensor]:
        """反量化 FP4 数据为 FP8 E4M3 格式，供 FlashInfer prefill 内核使用。"""
        ...

    @abstractmethod
    def compute_cell_size(self, head_num: int, head_dim: int, num_layers: int, kv_size: int) -> int:
        """每 token 内存占用估计（字节）。"""
        ...

    def load_scales_from_model(self, model_runner, sm_version: int = None) -> None:
        """从模型权重加载每层全局缩放（默认为无操作）。"""
        pass


class NVFP4KVMethod(FP4KVCacheQuantMethod):
    """NVFP4 双层缩放：全局 FP32 + 每块 FP8 E4M3，支持 SM100/SM120。"""
    name = "nvfp4"
    SCALE_BLOCK_SIZE = 16

    def __init__(self, num_layers: int, device: str, sm_version: int = 120):
        self.num_layers = num_layers
        self.device = device
        self.sm_version = sm_version
        # 每层全局缩放初始化为 1.0
        self.k_scales_gpu = torch.ones(num_layers, dtype=torch.float32, device=device)
        self.v_scales_gpu = torch.ones(num_layers, dtype=torch.float32, device=device)
```

```
    def needs_dequant_workspace(self) -> bool:
        # prefill 使用 FP8 反量化工作区；未来原生 FP4 内核可设为 False
        return True

    def needs_global_scale(self) -> bool:
        return True

    def load_scales_from_model(self, model_runner, sm_version: int = None) -> None:
        if sm_version is not None:
            self.sm_version = sm_version
        # 从模型权重读取全局缩放（具体实现略）
        ...
```

python/sglang/srt/layers/quantization/kvfp4_tensor.py

新增 NVFP4KVQuantizeUtil（FlashInfer 内核集成）和 BlockFP4KVQuantizeUtil（纯 PyTorch 块级量化），同时引入 FP4KVCacheRecipe 枚举统一 FP4 格式标识，是量化工具的核心实现。

```
# kvfp4_tensor.py — NVFP4 量化工具类，封装 FlashInfer 内核

class NVFP4KVQuantizeUtil:
    """
    NVFP4 量化/反量化工具。
    量化公式: x_fp4 * block_scale * global_scale ≈ x_bf16
    使用 FlashInfer nvfp4_kv_quantize (SM100+) 或 fp4_quantize (SM90 fallback)。
    """

    @staticmethod
    def quantize(tensor: torch.Tensor, global_scale: torch.Tensor) -> tuple[torch.Tensor, torch.
    Tensor, torch.Tensor]:
        """
        将 BF16/FP16 张量量化为 NVFP4 格式。
        输入形状: [B, M, N]；输出: (fp4_data [B, M, N/2], block_scales [B, M, N/16], global_scale)
        。
        """
        from sglang.srt.utils import is_sm90_supported, is_sm100_supported
        assert is_sm90_supported(), "NVFP4 量化需要 SM90+ GPU"
        b, m, n = tensor.shape
        tensor_2d = tensor.reshape(b * m, n)

        if isinstance(global_scale, (int, float)):
            global_scale = torch.tensor([global_scale], dtype=torch.float32, device=tensor.device)
        elif global_scale.dim() == 0:
            global_scale = global_scale.unsqueeze(0)

        # SM100+ 使用原生 PTX 内核
        if is_sm100_supported():
            try:
                from flashinfer import nvfp4_kv_quantize
```

```
            fp4_data = torch.empty(b * m, n // 2, dtype=torch.uint8, device=tensor.device)
            block_scales = torch.empty(b * m, n // 16, dtype=torch.float8_e4m3fn, device=
            tensor.device)
            nvfp4_kv_quantize(tensor_2d, global_scale, fp4_data, block_scales)
        except ImportError:
            raise RuntimeError("SM100+ 需要 FlashInfer 支持 nvfp4_kv_quantize")
    else:
        # SM90 fallback: fp4_quantize
        from flashinfer import fp4_quantize
        fp4_data, block_scales = fp4_quantize(tensor_2d, global_scale)

    return fp4_data.reshape(b, m, n // 2), block_scales.reshape(b, m, n // 16), global_scale

@staticmethod
def dequantize(fp4_data: torch.Tensor, block_scales: torch.Tensor, global_scale: torch.
Tensor,
               layer_id: int, dtype: torch.dtype = torch.bfloat16) -> torch.Tensor:
    """反量化 NVFP4 数据为指定精度（默认 BF16）。"""
    from sglang.srt.utils import is_sm100_supported
    assert is_sm100_supported(), "NVFP4 反量化需要 SM100+ GPU"
    from flashinfer import nvfp4_kv_dequantize
    return nvfp4_kv_dequantize(fp4_data, block_scales, global_scale, dtype)
```

评论区精华

- 抽象方法标记：review gemini-code-assist 建议将 compute_cell_size 标记为 @abstractmethod，samuellees 确认并修复（commit "Address PR review feedback"）。
- CUDA Graph 兼容性：b8zhong 询问反量化逻辑是否与分段 CUDA Graph 不兼容，samuellees 在 docstring 中注明“prefill-only，不在 CUDA graph 路径”，但后续又更新为“操作使用了 FlashInfer 内核和纯张量操作，因此是 CUDA graph 兼容的”。最终注释反映这些方法适用于 prefill，不打断 decode 的 CUDA graph。
- FlashInfer API 选择：samuellees 在自评中建议使用 fp4_kv_quantize 而不是 fp4_quantize，并在后续提交中改用 nvfp4_kv_quantize 配合显式 SM 版本检查。
- BlockFP4 块大小：DehuaTang 提问“为什么 block size 改为 16 而不是 OCP MXFP4 标准的 32”，samuellees 回复该问题不在该 PR 范围内（参考其他 PR #21954 的相关行）。
  - compute_cell_size 应标记为 @abstractmethod (correctness): 决议：应用建议，将 compute_cell_size 标记为 @abstractmethod，并在 NoneMethod（后移除）、NVFP4KVMethod、BlockFP4KVMethod 中实现。
  - CUDA Graph 兼容性处理 (correctness): 决议：代码注释最终明确量化操作仅用于 prefill，不参与 decode 的 CUDA Graph；但操作本身兼容。
  - 采用 nvfp4_kv_quantize 代替 fp4_quantize (performance): 决议：修改为 nvfp4_kv_quantize，配合显式 SM 版本检查，提高性能和正确性。
  - BlockFP4 块大小不符合 MXFP4 标准 (question): 决议：未解决，被标记为超出范围。

风险与影响

- 风险：
  1. 硬件依赖风险：NVFP4KVQuantizeUtil 要求 SM100+（主要通过 is_sm100_supported 和 is_sm90_supported 断言）。若部署在 SM90 以下的 GPU 上会直接报错，需确保调用路径已前置检查。
  2. FlashInfer 内核版本兼容性：依赖 nvfp4_kv_quantize 和 nvfp4_kv_dequantize，若 FlashInfer 版本未包含这些内核则导致 ImportError（当前代码通过 try/except fallback 到 PyTorch 实现，但只覆盖 SM90；SM100 无法 fallback）。
  3. 向后兼容别名：KVFP4QuantizeUtil = BlockFP4KVQuantizeUtil 保留在 kvfp4_tensor.py 中，其他模块（如 memory_pool.py）可能继续引用原名称，若未来移除会导致断裂。
  4. 未覆盖量化路径：该 PR 仅定义接口和单 token 操作，实际批量管理和解码集成在后续 PR 中，当前代码无法单独运行。
- 影响：
  - 用户影响：暂无直接影响，该 PR 为纯新增抽象层；后续 PR 完成后用户可通过配置 kv_cache_dtype="fp4_e2m1" 等启用 NVFP4 KV cache。
  - 系统影响：导入路径增加 fp4_kv_cache_quant_method，但不修改现有量化路径（如 FP8）。新增的 FP4KVCacheRecipe 枚举和注册表可被后续模块使用。
  - 团队影响：明确了“量化策略→池→后端”三层职责分离，降低后续功能合并复杂度和认知负荷。
  - 风险标记：硬件依赖（SM100+），新量化路径未在生产验证，FlashInfer 内核版本兼容性，向后兼容别名可能引入混淆

关联脉络

- 暂无明显关联 PR