

# PR #53132 完整报告

vllm-project/vllm

Support kimi k3 nvfp4 checkpoint

合并时间: 2026-08-21 13:59

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/53132>

## 执行摘要

- 一句话: 支持 Kimi K3 NVFP4 检查点加载与推理
- 推荐动作: 值得精读。重点读三处设计: `_compute_g1_scale_c` 对 SITU 激活的折叠语义 (非线性在 `scale` 折叠之前 / 之后决定了 `g1_scale_c` 的构成)、`FP8_PB_WO` 在 `linear method` 内部透明消化输出 `padding` 的思路、以及 KDA 层对 `block scale` 在 TP 下复制的启发式判断。这三个决策对后续支持其他 ModelOpt NVFP4 检查点 (如 DeepSeek V4 #52447) 有直接复用价值。

## 功能与动机

PR body 明确目的: Support running kimi k3 nvfp4 checkpoint [nvidia/Kimi-K3-NVFP4](#)。Kimi K3 的 NVFP4 检查点由 ModelOpt 导出, 其中 `FP8_PB_WO` 线性层输出宽度不保证 128 对齐、SITU 激活对 `g1_scale_c` 的折叠假设与现有实现不一致, 导致无法直接加载。社区成员 DamirLisak 也在评论中询问 DeepSeek V4 NVFP4 (#52447) 的同类支持, 说明 ModelOpt NVFP4 检查点兼容性是普遍诉求。

## 实现拆解

1. MoE 内核层 ([trtllm\\_nvfp4\\_moe.py](#)): 将原本在 `__init__` 与 `process_weights_after_loading` 中重复的 `g1_scale_c` 计算收拢为新方法 `_compute_g1_scale_c`, 并在 `__init__` 提前记录 `is_situ`。SITU 激活下非线性作用在 `g1_alphas` 之后, 因此只返回 `a2_gscale`, 避免把激活系数二次折叠进 `scale`。
2. 量化层 ([modelopt.py](#)): `ModelOptFp8PbWoLinearMethod` 对 `out_features` 非 128 整数倍由硬报错改为内部 `padding`: `create_weights` 计算 `output_padding` 并以物理宽度创建 `kernel`, `process_weights_after_loading` 用零填充补齐权重大小, `apply` 输出裁剪回 `logical_output_size` 再叠加 `bias`, 对上层透明; 同时把权重后处理委托从 `fp8_linear` 修正为 `w8a8_block_fp8_linear`。
3. KDA 层 ([kda.py](#)): `_KimiGDNMergedColumnParallelLinear` 的 `weight_loader/weight_loader_v2` 增加 `replicate_block_scale` 分支, 当参数是 `BlockQuantScaleParameter` 且加载张量的输出维度小于 `tp_size` 时按复制方式加载 (NVFP4 的 `block scale` 是全局张量, TP 下不能切分)。
4. 注意力层 ([mla\\_attention.py](#)): `_get_kv_b_proj_input_dtype` 对 FP8 权重新增分支: 若 `kv_b_proj` 的量化方法是 `ModelOptFp8PbWoLinearMethod`, 返回其 `input_dtype` (`FP8_PB_WO` 会在线性层内部动态量化 BF16/FP16 输入, 上游不需要再按 FP8 输入处理)。

## 5. 测试配套: tests/quantization/test\_modelopt.py 新增

test\_modelopt\_fp8\_pb\_wo\_hides\_output\_padding, 验证 padding 行为对调用方透明、bias 正确、输出连续; tests/kernels/moe/test\_trtllm\_nvfp4\_moe.py 注册 \_SituAndMulTest 并新增 situ 激活用例, 显式断言 SITU 时  $g1\_scale\_c == a2\_gscale$ 。

关键文件:

- vllm/model\_executor/layers/fused\_moe/experts/trtllm\_nvfp4\_moe.py (模块 专家内核; 类别 source; 类型 data-contract; 符号 `_compute_g1_scale_c`): 核心 MoE 内核适配: 新增 `_compute_g1_scale_c`, 修正 SITU 激活下 `g1_scale_c` 的折叠方式, 避免 NVFP4 检查点下激活系数被重复计入 scale。
- vllm/model\_executor/layers/quantization/modelopt.py (模块 量化层; 类别 source; 类型 core-logic; 符号 `create_weights`, `process_weights_after_loading`, `apply`): FP8\_PB\_WO 量化方法的通用增强: 输出宽度不满足 128 block 对齐时由报错改为 padding/ 裁剪, 使 nvidia/Kimi-K3-NVFP4 中非对齐线性层可加载。
- vllm/models/kimi\_k3/nvidia/kda.py (模块 门控注意力; 类别 source; 类型 data-contract; 符号 `_KimiGDNMergedColumnParallelLinear.weight_loader`, `_KimiGDNMergedColumnParallelLinear.weight_loader_v2`): KDA 权重加载在 TP 下的 block scale 复制修复, NVFP4 检查点的 BlockQuantScaleParameter 是全局张量, 不能按 TP 切分。
- vllm/model\_executor/layers/attention/mla\_attention.py (模块 潜在注意力; 类别 source; 类型 data-contract; 符号 `_get_kv_b_proj_input_dtype`): MLA 的 kv\_b\_proj 输入 dtype 判断需感知 FP8\_PB\_WO 动态量化策略, 否则 NVFP4 模型下 prefill 输入类型推断错误。
- tests/quantization/test\_modelopt.py (模块 量化测试; 类别 test; 类型 test-coverage; 符号 `test_modelopt_fp8_pb_wo_hides_output_padding`): 新增 test\_modelopt\_fp8\_pb\_wo\_hides\_output\_padding, 完整覆盖 padding 创建、零填充、kernel 调用与输出裁剪 +bias, 是 FP8\_PB\_WO 改动的主要回归保障。
- tests/kernels/moe/test\_trtllm\_nvfp4\_moe.py (模块 内核测试; 类别 test; 类型 test-coverage; 符号 `_SituAndMulTest`, `test_trtllm_fp4_moe_no_graph`): 为 TRTLLM NVFP4 MoE 内核新增 SITU 激活端到端用例, 并断言 SITU 下  $g1\_scale\_c == a2\_gscale$ , 锁定本次 scale 语义变更。

关键符号: `_compute_g1_scale_c`, `ModelOptFp8PbWoLinearMethod.create_weights`, `ModelOptFp8PbWoLinearMethod.process_weights_after_loading`, `ModelOptFp8PbWoLinearMethod.apply`, `_KimiGDNMergedColumnParallelLinear.weight_loader`, `_KimiGDNMergedColumnParallelLinear.weight_loader_v2`, `_get_kv_b_proj_input_dtype`, `test_modelopt_fp8_pb_wo_hides_output_padding`

## 关键源码片段

vllm/model\_executor/layers/fused\_moe/experts/trtllm\_nvfp4\_moe.py

核心 MoE 内核适配: 新增 `_compute_g1_scale_c`, 修正 SITU 激活下 `g1_scale_c` 的折叠方式, 避免 NVFP4 检查点下激活系数被重复计入 scale。

```

class TrtLlmNvFp4ExpertsBase:
    """NVFP4 TRTLLM-Gen MoE 内核，支持 modular / monolithic 两种接口。"""

    def __init__(
        self,
        moe_config: FusedMoEConfig,
        quant_config: FusedMoEQuantConfig,
        per_token_activation: bool = False,
    ):
        ...
        # 提前记录 SITU 激活；g1_scale_c 的折叠规则和 gemm 参数都依赖它
        self.is_situ = moe_config.activation == MoEActivation.SITU
        self.g1_scale_c = self._compute_g1_scale_c()
        ...

    def _compute_g1_scale_c(self) -> torch.Tensor:
        assert self.quant_config.g1_alphas is not None
        assert self.quant_config.a2_gscale is not None
        if not self.moe_config.is_act_and_mul:
            return self.quant_config.a2_gscale.clone()
        if self.is_situ:
            # SITU 的非线性激活作用在 g1_alphas 之后，g1_scale_c 里
            # 只应保留输出量化因子 a2_gscale，否则激活系数会被二次折叠
            return self.quant_config.a2_gscale.clone()
        # 常规路径：g1_alphas = a13_scale * w13_scale_2,
        # a2_gscale = 1 / a2_scale, 因此 g1_scale_c = g1_alphas * a2_gscale
        return self.quant_config.g1_alphas * self.quant_config.a2_gscale

    def process_weights_after_loading(self, layer: torch.nn.Module) -> None:
        layer.w13_weight_scale_2.data.mul_(layer.w13_input_scale)
        layer.w2_weight_scale_2.data.mul_(layer.w2_input_scale)
        # g1_alphas 已原地融合进权重 scale，需重算 g1_scale_c
        # 并注册为 layer 参数，让 EPLB 能随专家权重一起搬运
        g1_scale_c = self._compute_g1_scale_c()
        layer.register_parameter(
            "g1_scale_c", torch.nn.Parameter(g1_scale_c, requires_grad=False),
        )

```

## vllm/model\_executor/layers/quantization/modelopt.py

FP8\_PB\_WO 量化方法的通用增强：输出宽度不满足 128 block 对齐时由报错改为 padding/ 裁剪，使 [nvidia/Kimi-K3-NVFP4](#) 中非对齐线性层可加载。

```

class ModelOptFp8PbWoLinearMethod(LinearMethodBase):
    """ModelOpt FP8_PB_WO 线性方法；非 block 对齐的输出宽度会被 padding。"""

    _WEIGHT_BLOCK_SIZE: tuple[int, int] = (128, 128)

    def create_weights(
        self, layer, input_size_per_partition, output_partition_sizes,

```

```

input_size, output_size, params_dtype, **extra_weight_attrs,
):
    ...
    output_size_per_partition = sum(output_partition_sizes)
    block_n, block_k = self._WEIGHT_BLOCK_SIZE
    # out_features 不保证 128 对齐: 先算物理宽度, 内部透传 padding
    remainder = output_size_per_partition % block_n
    self.output_padding = 0 if remainder == 0 else block_n - remainder
    self.logical_output_size = output_size_per_partition
    physical_output_size = output_size_per_partition + self.output_padding

    weight = ModelWeightParameter(
        data=torch.empty(
            (output_size_per_partition, input_size_per_partition),
            dtype=torch.float8_e4m3fn,
        ),
        input_dim=1, output_dim=0, weight_loader=weight_loader,
    )
    layer.register_parameter("weight", weight)
    ...
    out_blks = physical_output_size // block_n
    in_blks = input_size_per_partition // block_k
    # 权重 scale 形状按物理宽度创建, 与 ModelOpt 导出的 4D 布局一致
    weight_scale = BlockQuantScaleParameter(
        data=torch.empty((out_blks, 1, in_blks, 1), dtype=torch.float32),
        input_dim=2, output_dim=0, weight_loader=weight_loader,
    )
    layer.register_parameter("weight_scale", weight_scale)
    self.w8a8_block_fp8_linear = init_fp8_linear_kernel(
        activation_quant_key=self.activation_quant_key,
        weight_quant_key=self.weight_quant_key,
        weight_shape=(physical_output_size, input_size_per_partition),
        input_dtype=self.input_dtype, out_dtype=self.out_dtype,
        module_name=self.__class__.__name__,
    )

def process_weights_after_loading(self, layer) -> None:
    weight = layer.weight.data
    if self.output_padding:
        # 尾部补零到物理宽度, 保证 kernel 的 128 对齐约束
        padded_weight = weight.new_zeros(
            self.logical_output_size + self.output_padding, weight.shape[1])
        padded_weight[:self.logical_output_size].copy_(weight)
        weight = padded_weight
    layer.weight = Parameter(weight, requires_grad=False)
    ...
    self.w8a8_block_fp8_linear.process_weights_after_loading(layer)

def apply(self, layer, x, bias=None) -> torch.Tensor:

```

```

# 由 kernel 内部完成动态量化与 block GEMM, bias 由本层处理
output = self.w8a8_block_fp8_linear.apply_weights(layer, x, None)
if self.output_padding:
    output = output[:, :self.logical_output_size]
if bias is not None:
    output = output + bias
return output.contiguous()

```

## vllm/models/kimi\_k3/nvidia/kda.py

KDA 权重加载在 TP 下的 block scale 复制修复, NVFP4 检查点的 **BlockQuantScaleParameter** 是全局张量, 不能按 TP 切分。

```

class _KimiGDNMergedColumnParallelLinear(MergedColumnParallelLinear):
    """Merged projection with one output replicated across TP ranks."""

    def weight_loader(self, param, loaded_weight, loaded_shard_id=None) -> None:
        tp_rank = self.tp_rank
        param_tp_rank = getattr(param, "tp_rank", None)
        # NVFP4 检查点的 block scale 是全局张量, TP 下每个 rank 都要完整副本;
        # 这里通过参数类型 + 输出维度小于 TP 大小来识别这类 scale
        replicate_block_scale = (
            isinstance(param, BlockQuantScaleParameter)
            and loaded_weight.shape[param.output_dim] < self.tp_size
        )
        if loaded_shard_id == self.replicated_shard_id or replicate_block_scale:
            self.tp_rank = 0
            if param_tp_rank is not None:
                param.tp_rank = 0
        try:
            super().weight_loader(param, loaded_weight, loaded_shard_id)
        finally:
            self.tp_rank = tp_rank
            if param_tp_rank is not None:
                param.tp_rank = param_tp_rank

    def weight_loader_v2(self, param, loaded_weight, loaded_shard_id=None) -> None:
        # v2 分支与 v1 逻辑一致, 同样识别 block scale 并复制加载
        tp_rank = self.tp_rank
        param_tp_rank = getattr(param, "tp_rank", None)
        replicate_block_scale = (
            isinstance(param, BlockQuantScaleParameter)
            and loaded_weight.shape[param.output_dim] < self.tp_size
        )
        if loaded_shard_id == self.replicated_shard_id or replicate_block_scale:
            self.tp_rank = 0
            if param_tp_rank is not None:
                param.tp_rank = 0
        try:
            super().weight_loader_v2(param, loaded_weight, loaded_shard_id)

```

```
finally:
    self.tp_rank = tp_rank
    if param_tp_rank is not None:
        param.tp_rank = param_tp_rank
```

## 评论区精华

PR 内没有实质 review 评论 ([claude\[bot\]](#) 因 fork PR 自动 review 被禁用, [zyongye](#) 直接 APPROVED)。唯一有价值的交互来自 issue 评论: [zyongye](#) 触发 `/ci run` 并跑通 Buildkite CI; [DamirLisak](#) 询问能否复用此改动支持 DeepSeek V4 NVFP4 (#52447), 该问题未在 PR 内回复, 但本 PR 的 FP8\_PB\_WO padding 与 SITU 处理为同类检查点提供了通用路径。

- DeepSeek V4 NVFP4 支持询问 (question): PR 内未回复; 该诉求指向 #52447, 本 PR 的 padding 与 SITU 处理为后续支持提供了通用基础。
- CI 触发与结果 (other): CI 通过后 PR 被合入 main。

## 风险与影响

- 风险:
  1. 量化语义变更 ([trtllm\\_nvfp4\\_moe.py](#)) : `g1_scale_c` 对 SITU 激活改走 `a2_gscale` 分支, 若其他模型也用 SITU + NVFP4 且依赖旧折叠语义, 精度会变化; 测试已补 SITU 用例, 但覆盖仍限于单模型。
  2. padding 透明化 ([modelopt.py](#)) : FP8\_PB\_WO 从报错改为静默 padding, apply 中裁剪 + bias 的先后顺序假设输出 padding 在尾部, 且测试只覆盖 TP=1 场景; TP>1 时物理宽度分配、EP 场景的权重搬运 (EPLB) 是否完全兼容尚缺端到端验证。
  3. TP 复制判断 ([kda.py](#)) : `replicate_block_scale` 用 `loaded_weight.shape[output_dim] < self.tp_size` 启发式识别, 理论上可能误伤真正小 shard 的普通参数, 但限定了 `BlockQuantScaleParameter` 类型, 风险可控。
  4. 惰性导入: `mha_attention.py` 在函数内导入 `ModelOptFp8PbWoLinearMethod`, 避免模块级循环依赖, 但若量化层异常, 错误会被延迟到运行期。
    - 影响: 对用户: Kimi K3 用户可直接 `vllm serve nvidia/Kimi-K3-NVFP4`, TP8/TEP8/DCP8 三种并行配置下 GSM8K 精度 0.9575-0.9606, 与官方精度相当。对系统: 改动横跨 MoE 内核、ModelOpt 量化层、KDA 权重加载与 MLA 注意力四个通用模块, 其中 FP8\_PB\_WO padding 和 SITU scale 语义是通用路径, 会同时影响其他使用 ModelOpt NVFP4/FP8\_PB\_WO 的模型 (为 DeepSeek V4 NVFP4 等铺路)。对团队: 为 NVFP4 检查点支持补齐了块对齐和激活折叠两个关键缺口, 测试覆盖同步到位。影响程度中高。
    - 风险标记: 核心路径变更, 量化语义变更, 跨模块影响, 并行场景覆盖不足

## 关联脉络

- PR #53053 [Kimi-K3] Extend GEMM-RS to GEMM-AR: 同属 Kimi K3 NVIDIA 实现目录, 且同样改动 `kda.py` 等 K3 内核文件, 是 K3 支持线路上紧邻的增强。
- PR #53043 [Rust Frontend] Fix Kimi K3 reasoning\_effort="none" handling: Kimi K3 支持链路的另一环 (前端参数处理), 与本次模型后端 NVFP4 支持互补。