

PR #55356 完整报告

vllm-project/vllm

[Kimi Perf] Group fp8 mla cahche insertion, 4~6x kernel level performance improvement for small batch

合并时间: 2026-09-11 23:16

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/55356>

执行摘要

- 一句话: Kimi K3 MLA 缓存插入支持 FP8 分组内核, 小 batch 提速约 6 倍
- 推荐动作: 值得精读。三条设计决策值得关注: (1) 用模板参数而非运行时分支承担 BF16/FP8 差异, hot loop 中 if constexpr 零开销; (2) 每层 cache 基址与 scale 都放进设备端 tensor, 一次 launch 内并行读取, 这是 5 层合并后还能继续扩展到更多层的关键; (3) process_weights_after_loading 的时序修复揭示了 vLLM 里 _k_scale 这类量化 scale 的定稿时机, 凡是要缓存量化参数的地方都应遵循同样的约束。建议读者重点比对 cache_kernels.cu 的派发宏与 host 校验顺序, 以及测试中 shared_slot_mapping 别名场景的覆盖方式; 若后续要在其它模型复用该内核, 需要先确认 _has_uniform_block_layout 这类前提在目标模型上是否成立。

功能与动机

Kimi K3 的 DSpark 草稿模型有 5 个 MLA draft layer, 每个 token 都要为这 5 层分别写入 context KV。PR body 描述旧路径是「CUDA launch 1: layer 0 / CUDA launch 2: layer 1 ……」, 即 5 次独立 launch; 目标是「combining 5 layers together, saving a lot of launch time」。由于线上 K3 服务普遍使用 fp8 KV cache, 而 base 侧已有的 concat_and_cache_mla_grouped 只支持 BF16 (schema 注释写明 bf16 only, 模型侧用 not is_quantized_kv_cache(...) 做前置判断), FP8 场景无法受益于分组写入, 因此本次把该内核扩展到 plain FP8, 并把每层 k_scale 一起下沉到设备端 tensor。

实现拆解

1. 变更入口与调用链: 调用方是 `vllm/models/kimi_k3/nvidia/dspark_mla.py::_precompute_fused_context_kv`, 跨层链路为 Python 绑定 `vllm/_custom_ops.py::concat_and_cache_mla_grouped` → torch schema (`csrc/libtorch_stable/ops.h` 声明 + `csrc/libtorch_stable/torch_bindings.cpp` 注册) → CUDA/HIP 实现 `csrc/libtorch_stable/cache_kernels.cu::concat_and_cache_mla_grouped`。
2. 内核模板化以承载 FP8: `concat_and_cache_mla_grouped_kernel` 由单模板参数改为 `<typename scalar_t, typename cache_t, Fp8KVCacheDataType kv_dt>`, cache 基址指针按 `cache_t` 重新解释 (BF16 为 `uint16_t`, FP8 为 `uint8_t`), 并在 copy lambda 内用 `if constexpr` 分流: kAuto 走原始 16-bit 直拷, FP8 走 `fp8::scaled_convert<cache_t, scalar_t, kv_dt>`。新增 `const float* kv_scales ([num_layers], BF16 传 nullptr)`, 按

layer_idx 取 scale, 避免 host 侧循环传参。

3. host 侧校验与派发: `concat_and_cache_mla_grouped` 新增 `std::optional<Tensor> kv_scales` 与 `const std::string& kv_cache_dtype`, 先做 dtype 白名单 (fp8/fp8_e4m3/fp8_e5m2/auto/bfloat16)、输入必须 BF16、`kv_cache_ptrs` 与 `slot_mapping` 必须 int64 的校验; FP8 分支再校验 `kv_scales` 的 dtype/numel/device/contiguity, 非 FP8 分支则要求 `kv_scales` 必须为空 (杜绝静默忽略)。ROCm 下用 `#ifdef USE_ROCM` 直接拒绝 fp8_e5m2, 最后按 dtype 派发到 kAuto、kFp8E5M2、kFp8E4M3 三个模板实例 (用 `LAUNCH_GROUPED_FP8` 宏收敛)。

4. op schema 与 Python 绑定: `ops.h` 签名与 `torch_bindings.cpp` 的 `ops.def` 增加 `Tensor? kv_scales=None, str kv_cache_dtype='auto'` 默认参数, 保持旧调用点二进制 / 源码兼容; `vllm/_custom_ops.py` 的包装函数同步加参并原样透传。

5. 模型侧接线与权重时序修复: `dspark_mla.py` 新增 `_GROUPED_KV_CACHE_DTYPES` 白名单; `_build_fused_context_kv_metadata` 中堆叠每层 `attn.k_scale.reshape()` 得到 `_context_kv_scales`; `_precompute_fused_context_kv` 的准入条件由 `not is_quantized_kv_cache(...)` 改为 `cache_dtype in _GROUPED_KV_CACHE_DTYPES and all(cl.kv_cache_dtype == cache_dtype ...)`, 并把 `_context_kv_scales`、`cache_dtype` 传入 `grouped op`。同时把 `_build_fused_context_kv_metadata()` 从 `load_weights` 末尾移到新 `hook process_weights_after_loading`, `load_weights` 只保留 `return loader.load_weights(...)`。

6. 测试与配置配套: `tests/kernels/attention/test_cache.py` 新增 `test_concat_and_cache_mla_grouped`, 参数化 `shared_slot_mapping` (真 / 假) 与 `kv_cache_dtype` (auto/fp8/fp8_e5m2), 用逐层 `ops.concat_and_cache_mla` 构造 reference 后以 `rtol=0, atol=0` 严格要求一致, 并断言 CPU scale 与非连续 scale 会报错; ROCm 下 skip fp8_e5m2。无新增配置项或部署改动。

关键文件:

- `csrc/libtorch_stable/cache_kernels.cu` (模块 缓存内核; 类别 source; 类型 core-logic; 符号 `concat_and_cache_mla_grouped_kernel`, `concat_and_cache_mla_grouped`, `LAUNCH_GROUPED_FP8`): 分组 MLA 缓存写入内核的核心改造: 模板参数扩展为, 新增 `kv_scales` 设备指针与 `fp8::scaled_convert` 写入分支, host 侧新增完整参数校验、ROCm fp8_e5m2 守卫与三分支派发, 是本 PR 性能收益的来源。
- `vllm/models/kimi_k3/nvidia/dspark_mla.py` (模块 K3 草稿模型; 类别 source; 类型 data-contract; 符号 `_GROUPED_KV_CACHE_DTYPES`, `_build_fused_context_kv_metadata`, `_precompute_fused_context_kv`, `process_weights_after_loading`): 模型侧接线: 新增 dtype 白名单与每层 `k_scale` 快照, 放宽 `grouped` 路径准入条件, 并按 review 意见把元数据构建迁移到 `process_weights_after_loading`, 属于影响正确性的 data-contract 变更。
- `csrc/libtorch_stable/torch_bindings.cpp` (模块 算子注册; 类别 source; 类型 entrypoint; 符号 `concat_and_cache_mla_grouped`): 注册 `concat_and_cache_mla_grouped` 的 torch schema, 新增 `kv_scales` 可选参数与 `kv_cache_dtype` 默认值, 是 Python 侧能传入每层 scale 的入口; 注释也由 `bf16 only` 改写为支持 FP8。

- `csrc/libtorch_stable/ops.h` (模块 算子声明; 类别 source; 类型 data-contract; 符号 `concat_and_cache_mla_grouped`) : C++ 声明层同步扩展签名 (`std::optional kv_scales + kv_cache_dtype`) , 保证 `libtorch_stable` 与 `.cu` 实现接口一致。
- `vllm/_custom_ops.py` (模块 算子绑定; 类别 source; 类型 data-contract; 符号 `concat_and_cache_mla_grouped`) : Python 绑定同步加参并透传, 是模型侧调用 `grouped` 算子的直接接口。
- `tests/kernels/attention/test_cache.py` (模块 内核测试; 类别 test; 类型 test-coverage; 符号 `test_concat_and_cache_mla_grouped, run_grouped`) : 新增 `test_concat_and_cache_mla_grouped`, 用逐层调用构造 reference 做 bit 级对照, 覆盖 `shared slot mapping`、`auto/fp8/fp8_e5m2` 与非法 `scale` 的报错路径, 并处理 `ROCm skip`。

关键符号: `concat_and_cache_mla_grouped_kernel`, `concat_and_cache_mla_grouped`, `_build_fused_context_kv_metadata`, `_precompute_fused_context_kv`, `process_weights_after_loading`, `test_concat_and_cache_mla_grouped`

关键源码片段

`csrc/libtorch_stable/cache_kernels.cu`

分组 MLA 缓存写入内核的核心改造: 模板参数扩展为 , 新增 `kv_scales` 设备指针与 `fp8::scaled_convert` 写入分支, `host` 侧新增完整参数校验、`ROCm fp8_e5m2` 守卫与三分支派发, 是本 PR 性能收益的来源。

```
// 分组版 concat_and_cache_mla: grid = (num_tokens, num_layers), 一次 launch 写完全部 draft layer.
```

```
// 旧实现只支持 BF16 (注释写着 bf16 only) , template 只有一个 scalar_t;
```

```
// 这里增加 cache_t 与 kv_dt, 让 bf16 直拷与 fp8 量化写共用同一份索引逻辑。
```

```
template <typename scalar_t, typename cache_t, Fp8KVCacheDataType kv_dt>
```

```
__global__ void concat_and_cache_mla_grouped_kernel(
```

```
    const scalar_t* __restrict__ kv_c, // [num_layers, num_tokens, kv_lora_rank]
```

```
    const scalar_t* __restrict__ k_pe, // [num_layers, num_tokens, pe_dim]
```

```
    const int64_t* __restrict__ kv_cache_ptrs, // [num_layers], 每层 cache 基址 (copy_blocks_
    kernel 同款指针数组)
```

```
    const float* __restrict__ kv_scales, // [num_layers], BF16 路径传 nullptr
```

```
    const int64_t* __restrict__ slot_mapping, // [num_layers, num_tokens]
```

```
    /* ... 各层 stride / block_stride / entry_stride / block_size / kv_lora_rank / pe_dim ... */) {
```

```
    const int64_t layer_idx = blockIdx.y; // 第二维 grid 即 layer, 天然跨层并行
```

```
    const int64_t token_idx = blockIdx.x;
```

```
    const int64_t slot_idx = slot_mapping[token_idx + layer_idx * slot_layer_stride];
```

```
    if (slot_idx < 0) return; // 占位 / padding slot 直接跳过
```

```
    const int64_t block_idx = slot_idx / block_size;
```

```
    const int64_t block_offset = slot_idx % block_size;
```

```
// cache_t 决定写入宽度: BF16 为 uint16_t, FP8 为 uint8_t;
```

```
// 由模板参数而非运行时分支决定, hot loop 里没有 dtype 判断开销。
```

```
cache_t* __restrict__ kv_cache =
```

```
    reinterpret_cast<cache_t*>(kv_cache_ptrs[layer_idx]);
```

```
float scale = 0.0f;
```

```

if constexpr (kv_dt != Fp8KVCacheDataType::kAuto) {
    // 每层一份 scale, 随 kernel 一起并行读取, 省掉 host 侧逐层传参
    scale = kv_scales[layer_idx];
}

auto copy = [&](const scalar_t* __restrict__ src, int64_t src_token_stride,
               int size, int offset) {
    for (int i = threadIdx.x; i < size; i += blockDim.x) {
        const int64_t src_idx = token_idx * src_token_stride + i;
        const int64_t dst_idx =
            block_idx * block_stride + block_offset * entry_stride + i + offset;
        if constexpr (kv_dt == Fp8KVCacheDataType::kAuto) {
            kv_cache[dst_idx] = src[src_idx]; // BF16: 原始 16-bit 直拷
        } else {
            // FP8: 复用单层 concat_and_cache_mla 同款 scaled_convert, 保证数值路径一致
            kv_cache[dst_idx] =
                fp8::scaled_convert<cache_t, scalar_t, kv_dt>(src[src_idx], scale);
        }
    }
};
// 省略: 随后以 copy(...) 分别写入 kv_c (offset 0) 与 k_pe (offset = kv_lora_rank) 两段
}

void concat_and_cache_mla_grouped(
    torch::stable::Tensor& kv_c, torch::stable::Tensor& k_pe,
    torch::stable::Tensor& kv_cache_ptrs, torch::stable::Tensor& slot_mapping,
    int64_t block_size, int64_t block_stride, int64_t entry_stride,
    std::optional<torch::stable::Tensor> kv_scales, // [num_layers] or None
    const std::string& kv_cache_dtype) {
    const bool use_fp8 = kv_cache_dtype == "fp8" || kv_cache_dtype == "fp8_e4m3" ||
        kv_cache_dtype == "fp8_e5m2";
#ifdef USE_ROCM
    // ROCm / HIP 侧没有 fp8_e5m2 的 KV cache 转换路径, 直接拒绝,
    // 避免调用方落到未支持的 HIP 分支后拿到难以定位的运行期错误。
    STD_TORCH_CHECK(kv_cache_dtype != "fp8_e5m2",
        "concat_and_cache_mla_grouped does not support fp8_e5m2 "
        "KV cache on ROCm");
#endif
    STD_TORCH_CHECK(use_fp8 || kv_cache_dtype == "auto" ||
        kv_cache_dtype == "bfloat16",
        "concat_and_cache_mla_grouped only supports BF16 and plain "
        "FP8 KV cache; got ", kv_cache_dtype);
    /* ... kv_c / k_pe 必须为 bf16; kv_cache_ptrs 与 slot_mapping 必须为 int64 ... */

    const float* kv_scales_ptr = nullptr;
    if (use_fp8) {
        // 这组检查来自 review 意见: 非法 scale 张量必须在 kernel launch 前被拦下
        STD_TORCH_CHECK(kv_scales.has_value(),
            "FP8 grouped cache insert requires kv_scales");
    }
}

```

```

STD_TORCH_CHECK(kv_scales->scalar_type() == torch::headeronly::ScalarType::Float,
    "kv_scales must be float32");
STD_TORCH_CHECK(kv_scales->numel() == num_layers,
    "kv_scales must contain one scale per layer");
STD_TORCH_CHECK(kv_scales->is_cuda() &&
    kv_scales->get_device_index() == kv_c.get_device_index(),
    "kv_scales must be on the same CUDA device as kv_c");
STD_TORCH_CHECK(kv_scales->is_contiguous(), "kv_scales must be contiguous");
kv_scales_ptr = kv_scales->const_data_ptr<float>();
} else {
    // BF16 路径不允许传 scale，杜绝“悄悄忽略参数”的调用错误
    STD_TORCH_CHECK(!kv_scales.has_value(),
        "BF16 grouped cache insert does not use kv_scales");
}

const dim3 grid(num_tokens, num_layers);
const dim3 block(std::min(kv_lora_rank, 512));

if (!use_fp8) {
    vllm::concat_and_cache_mla_grouped_kernel<uint16_t, uint16_t,
        vllm::Fp8KVCacheDataType::kAuto><<<grid, block, 0, stream>>>(
        /* kv_c / k_pe / kv_cache_ptrs / nullptr(scale) / slot_mapping / strides ... */);
    return; // BF16 与 FP8 走不同模板实例，避免为不支持的组合做无用实例化
}

#define LAUNCH_GROUPED_FP8(KV_DTYPE) \
    vllm::concat_and_cache_mla_grouped_kernel<__nv_bfloat16, uint8_t, KV_DTYPE> \
        <<<grid, block, 0, stream>>>(/* 参数：含 kv_scales_ptr ... */)

if (kv_cache_dtype == "fp8_e5m2") {
    LAUNCH_GROUPED_FP8(vllm::Fp8KVCacheDataType::kFp8E5M2);
} else {
    LAUNCH_GROUPED_FP8(vllm::Fp8KVCacheDataType::kFp8E4M3); // "fp8" 与 "fp8_e4m3"
    共用
}
#undef LAUNCH_GROUPED_FP8
}

```

vllm/models/kimi_k3/nvidia/dspark_mla.py

模型侧接线：新增 dtype 白名单与每层 k_scale 快照，放宽 grouped 路径准入条件，并按 review 意见把元数据构建迁移到 process_weights_after_loading，属于影响正确性的 data-contract 变更。

```

# 只有这些 KV cache dtype 能走 grouped 分组写入内核；
# auto / bfloat16 走 16-bit 直拷，fp8 系列走 scaled_convert 量化写。
_GROUPED_KV_CACHE_DTYPES = frozenset(
    {"auto", "bfloat16", "fp8", "fp8_e4m3", "fp8_e5m2"}
)

```

```

def _build_fused_context_kv_metadata(self) -> None:
    """Checkpoint 加载并完成权重后处理后，构建跨层元数据。"""
    attentions = [layer.self_attn for layer in self.layers]
    attn0 = attentions[0]
    # ... 校验各层 latent KV 几何一致，缓存 _num_context_layers / _context_kv_width 等 ...
    self._context_kv_scales: torch.Tensor | None = None
    # FP8 KV cache 时把每层 k_scale 堆成一个 [num_layers] 张量，
    # 一次传给 grouped kernel；在 __init__ 里取会读到尚未定稿的占位 scale。
    if attn0.kv_cache_dtype in _GROUPED_KV_CACHE_DTYPES and is_quantized_kv_cache(
        attn0.kv_cache_dtype
    ):
        self._context_kv_scales = torch.stack(
            [attn._k_scale.reshape(()) for attn in attentions]
        )

def _precompute_fused_context_kv(
    self,
    context_states: torch.Tensor,
    context_positions: torch.Tensor,
    context_slot_mapping: torch.Tensor | list[torch.Tensor | None] | None,
) -> None:
    # ... 一次 KV-only GEMM + 分组 RMSNorm + 分组 RoPE，得到 all_kv_c_normed 与 all_k_pe
    ...
    if context_slot_mapping is None:
        return

    cache_layers = [layer.self_attn for layer in self.layers]
    cache_dtype = cache_layers[0].kv_cache_dtype
    if (
        # 旧条件是 not is_quantized_kv_cache(...)，即 FP8 直接被排除；
        # 现在改为白名单 + 所有层 dtype 一致，才能安全共用 pointer/scale 数组。
        cache_dtype in _GROUPED_KV_CACHE_DTYPES
        and all(cl.kv_cache_dtype == cache_dtype for cl in cache_layers)
        and self._has_uniform_block_layout(cache_layers)
        and (
            isinstance(context_slot_mapping, torch.Tensor)
            or all(s is not None for s in context_slot_mapping)
        )
    ):
        # ... 把单张 slot_mapping 广播，或把多层别名张量堆成 [num_layers, num_tokens] ...
        ref_cache = cache_layers[0].kv_cache
        ops.concat_and_cache_mla_grouped(
            all_kv_c_normed,
            all_k_pe.squeeze(2),
            self._get_context_kv_cache_ptrs(cache_layers),
            slot_mapping,
            ref_cache.size(1),
            ref_cache.stride(0),
            ref_cache.stride(1),

```

```

        self._context_kv_scales, # FP8 才有值, BF16 传 None
        cache_dtype,
    )
    return

# 回退路径: dtype/layout 前提不满足时仍逐层插入, 保证正确性优先
for layer_idx, layer in enumerate(self.layers):
    slot_mapping = (
        context_slot_mapping[layer_idx]
        if isinstance(context_slot_mapping, (list, tuple))
        else context_slot_mapping
    )
    if slot_mapping is None:
        continue
    attn = layer.self_attn
    attn.impl.do_kv_cache_update(
        all_kv_c_normed[layer_idx], all_k_pe[layer_idx], attn.kv_cache,
        slot_mapping, attn.kv_cache_dtype, attn._k_scale,
    )

def load_weights(self, weights: Iterable[tuple[str, torch.Tensor]]) -> set[str]:
    loader = AutoWeightsLoader(self)
    weights = _duplicate_context_kv_weights(weights, len(self.model.layers))
    # 这里不再构建元数据: _k_scale 要等量化层的权重后处理结束才定稿
    return loader.load_weights(weights, mapper=self.hf_to_vllm_mapper)

def process_weights_after_loading(self) -> None:
    # 权重后处理完成后才做元数据构建 (含 k_scale 快照) ,
    # 对应 review 里 ZJY0516 追问的时序问题, commit 1328ae4 "fix weights order".
    self.model._build_fused_context_kv_metadata()

```

tests/kernels/attention/test_cache.py

新增 test_concat_and_cache_mla_grouped, 用逐层调用构造 reference 做 bit 级对照, 覆盖 shared slot mapping、auto/fp8/fp8_e5m2 与非法 scale 的报错路径, 并处理 ROCm skip。

```

@pytest.mark.parametrize("device", CUDA_DEVICES)
@pytest.mark.parametrize("shared_slot_mapping", [False, True])
@pytest.mark.parametrize("kv_cache_dtype", ["auto", "fp8", "fp8_e5m2"])
@torch.inference_mode()
def test_concat_and_cache_mla_grouped(
    device: str, shared_slot_mapping: bool, kv_cache_dtype: str
) -> None:
    # ROCm / HIP 没有 fp8_e5m2 的 KV cache 支持, CI 曾在此组合上失败
    if kv_cache_dtype == "fp8_e5m2" and current_platform.is_rocm():
        pytest.skip("fp8_e5m2 KV cache is not supported on ROCm/HIP")

    # ... 构造 5 层 kv_c / k_pe / kv_caches, block_size=16、8 个 block、42 个 token ...
    scales = torch.linspace(0.05, 0.25, num_layers, dtype=torch.float32)

```

```

slot_mapping = torch.stack(
    [torch.randperm(total_slots)[:num_tokens] for _ in range(num_layers)]
)
slot_mapping[:, -1] = -1 # 保留一个无效 slot, 验证 kernel 里的提前返回分支
if shared_slot_mapping:
    slot_mapping = slot_mapping[:, 1].expand(num_layers, -1) # 各层别名同一份 slot

# reference: 逐层调用单层 kernel, 即 grouped 版本要等价替换的路径
for layer_idx in range(num_layers):
    ops.concat_and_cache_mla(
        kv_c[layer_idx], k_pe[layer_idx], reference[layer_idx],
        slot_mapping[layer_idx], kv_cache_dtype, scales[layer_idx],
    )

ref_cache = kv_caches[0]
cache_ptrs = torch.tensor(
    [kv_caches[layer_idx].data_ptr() for layer_idx in range(num_layers)],
    dtype=torch.int64,
)

def run_grouped(kv_scales: torch.Tensor | None) -> None:
    ops.concat_and_cache_mla_grouped(
        kv_c, k_pe, cache_ptrs, slot_mapping,
        ref_cache.size(1), ref_cache.stride(0), ref_cache.stride(1),
        kv_scales, kv_cache_dtype,
    )

run_grouped(None if kv_cache_dtype == "auto" else scales)

# BF16 与 FP8 都必须与逐层结果完全一致: rtol=0、atol=0
torch.testing.assert_close(kv_caches, reference, rtol=0, atol=0)

if kv_cache_dtype == "fp8" and not shared_slot_mapping:
    # 回归覆盖 coderabbit 要求的两类非法 scale: 跨 device 与非连续
    noncontiguous_scales = torch.ones(num_layers, 2)[: , 0]
    assert not noncontiguous_scales.is_contiguous()
    for invalid_scales, error in (
        (scales.cpu(), "same CUDA device"),
        (noncontiguous_scales, "must be contiguous"),
    ):
        with pytest.raises(RuntimeError, match=error):
            run_grouped(invalid_scales)

```

评论区精华

讨论集中在三处。(1) correctness: ZJY0516 在 [dspark_mla.py:240](#) 追问 [_context_kv_scales](#) 「Could you confirm it will not change after model loading and weight processing?」, 作者回 「Nice catch, fixed the issue」, 随即把元数据构建从

`load_weights` 迁移到 `process_weights_after_loading` (commit 1328ae4 fix weights order)。
(2) security/correctness: coderabbitai[bot] 在 `cache_kernels.cu` 提 Major 稳定性问题，要求在取 `kv_scales_ptr` 前校验 contiguity 与 device，并补 CPU / 非连续 scale 的回归覆盖——最终落成 host 侧四条 `STD_TORCH_CHECK` 加测试里的 `pytest.raises(..., match=...)`。

(3) 平台兼容: ZJY0516 贴出 AMD CI 失败 `Unsupported data type of kv cache: fp8_e5m2`，作者加 `#ifdef USE_ROCM` 守卫并 skip 该参数组合 (commit 19347c8 fix rocm test)，ZJY0516 最终 APPROVED 并注明「LGTM if the AMD failures get fixed」。

- 量化 scale 快照的构建时机 (`load_weights` vs 权重后处理) (correctness): 作者回复「Nice catch, fixed the issue」，将 `_build_fused_context_kv_metadata()` 从 `load_weights` 移到新的 `process_weights_after_loading` hook (commit 1328ae4 “fix weights order”)，确保快照在量化后处理结束后生成。
- `kv_scales` 的 device 与 contiguity 校验缺失 (correctness): host 侧补上 `float32`、`numel == num_layers`、`is_cuda()` + `get_device_index` 一致、`is_contiguous()` 四条 `STD_TORCH_CHECK`；测试中新增 `pytest.raises(RuntimeError, match="same CUDA device" / "must be contiguous")` 覆盖，并按“BF16 路径禁止传 scale”对称约束。
- ROCm 上 `fp8_e5m2` KV cache 不受支持导致 CI 失败 (testing): 作者在 host 侧加 `#ifdef USE_ROCM` 守卫直接拒绝 `fp8_e5m2`，并在测试中按平台 skip 该参数组合 (commit 19347c8 “fix rocm test”)；ZJY0516 随后 APPROVED，留言「LGTM if the AMD failures get fixed」。

风险与影响

- 风险：数值一致性：FP8 分组路径必须与逐层 `concat_and_cache_mla` bit 级一致，测试用 `rtol=0`, `atol=0` 覆盖，但形状固定 (5 层、42 token、`block_size 16`、`kv_lora_rank 512`、`rope dim 64`)，未覆盖不同 `block_size`、非 512 latent 或 `fp8_e4m3` 显式字符串输入。
scale 快照时效：`_context_kv_scales` 是 `torch.stack` 出的值快照，修复后依赖 `process_weights_after_loading` 在量化后处理之后执行；若未来有运行时改写 `_k_scale` 的路径 (动态量化、KV cache 重配)，缓存会静默过期而不报错，属于隐性数值风险。
平台分支：`#ifdef USE_ROCM` 仅拒绝 `fp8_e5m2`，`fp8/fp8_e4m3` 仍走 HIP 编译的 `fp8::scaled_convert`，本次修改是被 CI 失败驱动补上的，说明该路径此前未被 AMD 覆盖充分。
性能边界：PR 自带基准显示 8192 token 时 grouped 与逐层调用基本持平 (0.99x)，16384 token 时 1.04x；模型侧没有按 token 数选择路径，大 batch 下会多出 `grid.y` 的层维度调度开销，虽未见回退但收益归零。
内核入参契约：FP8 分支把 `kv_c/k_pe` reinterpret 为 `__nv_bfloat16*`，依赖前置的 `bf16 dtype` 校验不被绕过。
- 影响：对使用者：Kimi K3 系列在 `fp8` KV cache 下的小 batch (1~2048 token) 服务直接受益，context KV 插入从约 100us 降到约 16us，且该开销发生在每个 draft/verify 步，低延迟在线推理的固定成本明显下降；大 batch 场景无明显变化。对系统：新增三条 kernel 模板实例 (`kAuto` / `kFp8E4M3` / `kFp8E5M2`)，编译时间与二进制体积小幅增加，其它模型路径不受影响，op schema 使用默认参数保持向后兼容。对团队：确立「跨层 grouped kernel + 每层参数走设备端 tensor (指针数组 + scale 数组)」的复用模式，为其他 MLA/MHA 变体的多层 cache 写入提供模板；同时强化了「量化 scale 只能在 `process_weights_after_loading` 之后取快照」这一时序约束。

- 风险标记：内核模板扩展，FP8 数值路径变更，量化 scale 快照时效，大 batch 收益归零，ROCm 兼容分支

关联脉络

- PR #56433 [ROCm][Bugfix] Fix AITER preshuffled FP8 block-scale kernel: 同样是 FP8 + MLA + ROCm 的数值路径修复 (aiter.py、deepseek_v4/amd、mla_attention.py)，与本 PR 新增的 FP8 内核分支和 ROCm 守卫属于同一技术域，可对照理解不同 FP8 实现在 NVIDIA/AMD 上的行为差异。
- PR #56138 [Bugfix] Pin EPLB and MLA host-to-device transfer buffers: 同样针对 MLA 路径上额外的 host 侧拷贝 / 隐式同步开销做修复并补回归测试，方法论与本 PR「合并多次 launch、把每层参数下沉到设备端」一致，可视为同一类性能治理的延续。
- PR #56214 [Model] Support DeepSeek-V4.1-Flash: 该 PR 触及 vllm/v1/attention/backends/mla/indexer.py 与 vllm/v1/core/kv_cache_utils.py，同属 MLA KV cache 基础设施演进；理解 grouping 写入时对 cache 指针 / 块布局的假设需要这部分背景。
- PR #56447 [Bugfix] Fix GLM-OCR MTP position masking during CUDA graph capture: 同样面向 MTP / 草稿模型这类小 batch 投机解码路径的内核级问题，与 Kimi K3 DSpark 草稿层的 context KV 写入场景相近，可对比小 batch 下 kernel launch 成本的处理方式。