

PR #38160 完整报告

sgl-project/sglang

[Feature] Support BF16 and batch-invariant inference with DeepEP v2

合并时间: 2026-09-16 05:30

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/38160>

执行摘要

- 一句话: DeepEP v2 支持 BF16 与批量不变的确定性推理
- 推荐动作: 值得精读。这是一个「通信集合正确性工程」的完整样本: 如何把 rank 相关的浮点求和变成 rank 无关、如何用配置校验避免静默错误、如何用 mock 观察 wire 格式、如何设计多 GPU 位级回归。重点看三个设计决策: `create_moe_dispatcher` 把 `quant_method` 传入所有后端分支 (当前仅 `deepep_v2` 消费, 未来可复用); `wire dtype` 跟随时 `checkpoint` 而非自由开关; 确定性 `reduce-scatter` 用 `input.clone()` 保持调用契约的细节。若团队正在做确定性推理或多卡一致性治理, 本 PR 的测试方法论 (4 卡批量 × 目标 rank 不变性 + `atol=0, rtol=0` 断言) 可直接借鉴。

功能与动机

DeepEP v2 此前只接受 128x128 blockwise FP8 专家, BF16 `checkpoint` 在层初始化就被拒绝; 同时 #35758 给 `_fwd_kernel_ep_scatter_2` 增加 `expert_start/num_experts` 后未同步更新 #35634 引入的 `ep_scatter_from_psum` 调用者, 导致 `prefill` 阶段报 `TypeError: dynamic_func() missing 2 required positional arguments`。更隐蔽的问题是: TP `reduce-scatter` 的浮点求和顺序取决于接收 rank, PR 描述指出「As a serving batch grows, a token can move to another rank and enter the first MoE layer with different bits」, 即使开启确定性推理, token 的位级结果仍随批次变化——`collective` 回归显示未修复前 8,192 个 BF16 元素中有 2,550 个 (31.1%) 在不同目标 rank 上结果不同。

实现拆解

实现按四步推进:

1. 修复 contiguous scatter 缺参: 在 `python/sglang/kernels/ops/moe/ep_moe_kernels.py` 的 `ep_scatter_from_psum` 调用处补传 `expert_start=0` 和本地专家数, 因为 DeepEP v2 `dispatch` 已经 `rebase` 了接收到的专家 ID, 全局索引不需要再次翻译; 代码路径是 `deepep_v2` 的 `prefill permute` (`_fwd_kernel_ep_scatter_2` 的 v2 调用者)。配套新增 `test/registered/kernel/moe/test_deepep_v2_contig_scatter.py`, 用 BF16 无 scale 与 FP8 带 scale 两组 CUDA 用例锁定行为, 两个用例在 PR base 上均失败。
2. 建立 wire dtype 决策链: `python/sglang/srt/layers/moe/fused_moe_triton/layer.py` 的 `create_moe_dispatcher` 新增 `quant_method` 参数, 先经 `_deepep_v2_experts_are_fp8` (`not isinstance(quant_method, UnquantizedFusedMoEMethod)`) 判断专家格式, 再调用 `python/sglang/srt/layers/moe/utils.py` 新增的

`get_deepest_v2_dispatcher_output_dtype` 解析 `--deepest-dispatcher-output-dtype`: 显式 flag 与 checkpoint 矛盾时抛 `ValueError`, 避免漏传参数静默选错 FP8;
`_validate_deepest_v2_quant_method` 同时放行 `UnquantizedFusedMoEMethod`.
`python/sglang/srt/layers/moe/token_dispatcher/deepest_v2.py` 的 `_DeepEPv2Impl` 与 `DeepEPv2Dispatcher` 构造函数新增 `use_fp8_dispatch`, `_get_buffer` 把它透传给 `ElasticBuffer`, `dispatch` 中 BF16 分支直接透传激活、不分配 scale;
`python/sglang/srt/layers/moe/moe_runner/deep_gemm.py` 的 `pre_permute_deepest_v2_to_deep_gemm` 接受「无 scale 的 BF16 输出」并跳过 `input_tensor_scale` 分配。

3. 实现确定性 reduce-scatter: `python/sglang/srt/distributed/parallel_state.py` 的 `reduce_scatter_tensor` 在 `SGLANG_ENABLE_DETERMINISTIC_INFERENCE` 下改为 `all_reduce(input.clone())` 后按 `rank_in_group` 本地切片; 新增 `_deterministic_reduce_scatterv` 用前缀和 offset 支持不等长分片, `reduce_scatterv` 同步接入。克隆输入是为了保持调用契约——部分调用者把输出别名到输入切片。同时移除 `python/sglang/srt/arg_groups/moe_hook.py` 中 DeepEP v2 与确定性推理互斥的拒绝逻辑; `ElasticBuffer receive sorting` 保持关闭, 因为专家计算行独立、combine 走 slot 指针, 且 sort 的动态索引与 decode CUDA graph 捕获不兼容。
4. 测试与 CI 配套: 新增 `test/registered/unit/layers/moe/test_deepest_v2_wire_dtype.py` (CPU mock, 观察 `ElasticBuffer` 收到的 wire 格式)、`test/registered/kernel/communication/test_deterministic_reduce_scatter.py` (4 卡回归, 覆盖 BF16/FP16/FP32 × eager/CUDA graph, 按生产 NCCL 通道数配置)、`test/manual/ep/test_deepest_v2_deterministic.py` (人工 logprob 探针, 校验服务器开启确定性且禁用 radix cache); `test/registered/unit/distributed/test_parallel_state.py` 覆盖确定性路径的输入保持与 rank 分片选择, `test/registered/unit/server_args/test_server_args.py` 把 `deterministic + deepest_v2` 从拒绝翻转为接受, `test/registered/unit/layers/moe/test_hpc_ops_runner_guard.py` 覆盖新量化契约。

关键文件:

- `python/sglang/srt/layers/moe/fused_moe_triton/layer.py` (模块 专家层; 类别 source; 类型 core-logic; 符号 `create_moe_dispatcher`, `_deepest_v2_experts_are_fp8`, `_validate_deepest_v2_quant_method`): `dispatcher` 装配入口: `create_moe_dispatcher` 新增 `quant_method` 参数并按专家格式解析 `use_fp8_dispatch`; `_validate_deepest_v2_quant_method` 放行 BF16 专家, 是 BF16 支持的总开关。
- `python/sglang/srt/distributed/parallel_state.py` (模块 通信层; 类别 source; 类型 core-logic; 符号 `reduce_scatter_tensor`, `reduce_scatterv`, `_deterministic_reduce_scatterv`): 确定性推理的核心修复: `reduce_scatter_tensor` 与 `reduce_scatterv` 在确定性开关下改为 `all-reduce` + 本地切片, 消除求和对接收 rank 的依赖; 属于全局分布式通信层路径。
- `python/sglang/srt/layers/moe/token_dispatcher/deepest_v2.py` (模块 专家分发; 类别 source; 类型 core-logic; 符号 `DeepEPv2Dispatcher.init`, `_DeepEPv2Impl.init`, `_DeepEPv2Impl._get_buffer`, `_DeepEPv2Impl.dispatch`): BF16 wire 的实际执行点: `use_fp8_dispatch` 透传到 `ElasticBuffer`, `dispatch` 中跳过激活量化, 是 BF16 通信的核心逻辑。

- python/sglang/srt/layers/moe/utils.py (模块 专家工具; 类别 source; 类型 core-logic; 符号 get_deepep_v2_dispatcher_output_dtype) : 新增 get_deepep_v2_dispatcher_output_dtype: 解析 --deepep-dispatcher-output-dtype 并在与 checkpoint 矛盾时抛错, 是防静电错误的配置校验点。
- python/sglang/srt/layers/moe/moe_runner/deep_gemm.py (模块 内核适配; 类别 source; 类型 core-logic; 符号 pre_permute_deepep_v2_to_deep_gemm) : DeepGEMM 适配器接受无 scale 的 BF16 dispatch 输出, 并跳过激活 scale 分配; 同时保留 FP8 契约。
- python/sglang/kernels/ops/moe/ep_moe_kernels.py (模块 内核封装; 类别 infra; 类型 bugfix; 符号 ep_scatter_from_psum) : 修复 ep_scatter_from_psum 缺参的关键 bugfix 位置: 补传 expert_start=0 与本地专家数。
- test/registered/unit/layers/moe/test_deepep_v2_wire_dtype.py (模块 格式测试; 类别 test; 类型 test-coverage; 符号 TestDeepEPv2WireDtype, TestDeepEPv2DispatcherOutputDtype, _FakeBuffer, _fake_quant) : CPU-only wire 格式测试: 用 _FakeBuffer mock 观察 BF16/FP8 两种 wire 输出及 ElasticBuffer 布局选择, 覆盖 dtype 解析的矛盾拒绝逻辑。
- test/registered/kernel/moe/test_deepep_v2_contig_scatter.py (模块 内核测试; 类别 test; 类型 test-coverage; 符号 TestDeepEPv2ContigScatter, _run, _check, test_scatter_bf16_without_scales) : CUDA scatter 回归: BF16 无 scale 与 FP8 带 scale 两个用例在 PR base 上均失败, 锁定缺参修复。
- test/registered/kernel/communication/test_deterministic_reduce_scatter.py (模块 确定性测试; 类别 test; 类型 test-coverage; 符号 group, test_batch_and_destination_invariance) : 4 卡集体回归: 验证 batch size 与目标 rank 变化时输出位级一致, 覆盖三种 dtype 与 CUDA graph 捕获场景, 是确定性修复的最高价值测试。
- test/registered/unit/distributed/test_parallel_state.py (模块 通信测试; 类别 test; 类型 test-coverage; 符号 test_deterministic_reduce_scatter_preserves_input_and_selects_rank_shard, test_deterministic_reduce_scatterv_selects_rank_shard, test_nondeterministic_reduce_scatter_keeps_native_path) : CPU 单测覆盖确定性 reduce-scatter 的输入保持、rank 分片选择及非确定性 native 路径不被破坏。

关键符号: create_moe_dispatcher, _deepep_v2_experts_are_fp8, _validate_deepep_v2_quant_method, get_deepep_v2_dispatcher_output_dtype, DeepEPv2Dispatcher.init, _DeepEPv2Impl.init, _DeepEPv2Impl._get_buffer, _DeepEPv2Impl.dispatch, pre_permute_deepep_v2_to_deep_gemm, reduce_scatter_tensor, reduce_scatterv, _deterministic_reduce_scatterv, ep_scatter_from_psum

关键源码片段

python/sglang/srt/layers/moe/fused_moe_triton/layer.py

dispatcher 装配入口: create_moe_dispatcher 新增 quant_method 参数并按专家格式解析 use_fp8_dispatch; _validate_deepep_v2_quant_method 放行 BF16 专家, 是 BF16 支持的

总开关。

```
def create_moe_dispatcher(
    moe_runner_config: MoeRunnerConfig,
    quant_method: FusedMoEMethodBase,
) -> BaseDispatcher:
    a2a_backend = get_moe_a2a_backend()
    # 其他后端分支省略: none / megamoe / deepEP v1 / flashinfer 等保持原样。
    if a2a_backend.is_deepEP_v2():
        # wire dtype 必须跟随 expert 权重量化方式: DeepEP v2 把接收缓冲
        # 直接交给 DeepGEMM 分组 GEMM, 量化方式不一致会在 GEMM 入口出错。
        output_dtype = get_deepEP_v2_dispatcher_output_dtype(
            _deepEP_v2_experts_are_fp8(quant_method)
        )
    return DeepEPv2Dispatcher(
        group=get_tp_group().device_group,
        router_topk=moe_runner_config.top_k,
        num_experts=moe_runner_config.num_experts,
        num_local_experts=moe_runner_config.num_local_experts,
        hidden_size=moe_runner_config.hidden_size,
        params_dtype=moe_runner_config.params_dtype,
        use_fp8_dispatch=output_dtype is DispatcherOutputDtype.FP8,
    )
```

```
def _deepEP_v2_experts_are_fp8(quant_method) -> bool:
    # 除未量化 BF16 (UnquantizedFusedMoEMethod) 外, 当前 deepEP_v2
    # 支持的其余量化方法都是 blockwise FP8。
    return not isinstance(quant_method, UnquantizedFusedMoEMethod)
```

python/slang/srt/distributed/parallel_state.py

确定性推理的核心修复: `reduce_scatter_tensor` 与 `reduce_scatterv` 在确定性开关下改为 all-reduce + 本地切片, 消除求和对接收 rank 的依赖; 属于全局分布式通信层路径。

```
def reduce_scatter_tensor(self, output: torch.Tensor, input: torch.Tensor):
    if envs.SGLANG_ENABLE_DETERMINISTIC_INFERENCE.get():
        assert input.numel() == output.numel() * self.world_size
        # 原生 reduce-scatter 的浮点求和顺序由接收 rank 决定: 同一条 token
        # 随批次增长换到其他 rank 后, 进入第一个 MoE 层时已是不同位模式。
        # all-reduce 让所有 rank 拿到同一归约结果, 再做本地切片,
        # 求和的 rank 依赖就此消除; 克隆 input 保持调用契约,
        # 兼容输出别名到输入切片的调用者。
        reduced = self.all_reduce(input.clone())
        output.copy_(
            reduced.reshape(-1)
            .narrow(0, self.rank_in_group * output.numel(), output.numel())
            .view_as(output)
        )
    elif _is_npu or _is_cpu:
```

```

        # TODO: add optimized reduce_scatter_tensor kernel for cpu
        self._reduce_scatter_tensor(output, input)
    elif self._maybe_aiter_reduce_scatter(output, input):
        return
    else:
        reg_reduce_scatter_tensor(output, input, group_name=self.unique_name)

def reduce_scatterv(
    self,
    input_: torch.Tensor,
    output: Optional[torch.Tensor] = None,
    sizes: Optional[List[int]] = None,
) -> torch.Tensor:
    world_size = self.world_size
    pynccl_comm = self.pynccl_comm

    # reduce_scatterv 不是边角路径: DP attention (attn_dp_size > 1) 时
    # should_use_dp_reduce_scatterv() 会把 combine 路由到这里。
    if envs.SGLANG_ENABLE_DETERMINISTIC_INFERENCE.get():
        return self._deterministic_reduce_scatterv(input_, output, sizes)

    with pynccl_comm.change_state(enable=True):
        assert pynccl_comm is not None and not pynccl_comm.disabled, (
            "pynccl is required for reduce_scatterv"
        )
        if sizes is not None:
            assert len(sizes) == world_size
            assert input_.shape[0] == sum(sizes)
            chunk_size = sizes[self.rank_in_group]
        else:
            assert input_.shape[0] % world_size == 0
            chunk_size = input_.shape[0] // world_size
        output_shape = (chunk_size,) + input_.shape[1:]
        if output is None:
            output = torch.empty(
                output_shape, dtype=input_.dtype, device=input_.device
            )
        else:
            assert output.shape == output_shape
        pynccl_comm.reduce_scatter(output, input_, sizes=sizes)
    return output

def _deterministic_reduce_scatterv(
    self,
    input_: torch.Tensor,
    output: Optional[torch.Tensor],
    sizes: Optional[List[int]],

```

```

) -> torch.Tensor:
    # 与 reduce_scatter_tensor 同理，但支持不等长分片：offset 是前缀和。
    if sizes is not None:
        assert len(sizes) == self.world_size
        assert input_.shape[0] == sum(sizes)
        chunk_size = sizes[self.rank_in_group]
        offset = sum(sizes[: self.rank_in_group])
    else:
        assert input_.shape[0] % self.world_size == 0
        chunk_size = input_.shape[0] // self.world_size
        offset = chunk_size * self.rank_in_group
    output_shape = (chunk_size,) + input_.shape[1:]
    if output is None:
        output = torch.empty(output_shape, dtype=input_.dtype, device=input_.device)
    else:
        assert output.shape == output_shape
    # 克隆输入：即使 all_reduce 原地改写参数，也不破坏调用方的 input。
    reduced = self.all_reduce(input_.clone())
    output.copy_(reduced.narrow(0, offset, chunk_size))
    return output

```

python/sglang/srt/layers/moe/token_dispatcher/deeep_v2.py

BF16 wire 的实际执行点：use_fp8_dispatch 透传到 ElasticBuffer，dispatch 中跳过激活量化，是 BF16 通信的核心逻辑。

```

class _DeepEPv2Impl:
    def __init__(
        self,
        group,
        router_topk: int,
        num_experts: int,
        num_local_experts: int,
        hidden_size: int,
        scale_format: DeepEPv2Fp8ScaleFormat,
        num_max_dispatch_tokens_per_rank: int,
        use_fp8_dispatch: bool,
    ):
        self.group = group
        self.router_topk = router_topk
        self.num_experts = num_experts
        self.num_local_experts = num_local_experts
        self.hidden_size = hidden_size
        self.scale_format = scale_format
        self.num_max_dispatch_tokens_per_rank = num_max_dispatch_tokens_per_rank
        self.use_fp8_dispatch = use_fp8_dispatch

    def _get_buffer(self) -> ElasticBuffer:
        # ElasticBuffer 的单例缓存键包含 use_fp8_dispatch: BF16 与 FP8 层
        # 交错的 checkpoint 会在层切换处触发集体缓冲区重建，需要单独的生命周期覆盖。

```

```

return ElasticBuffer(
    self.rank,
    self.hidden_size,
    self.router_topk,
    self.num_max_dispatch_tokens_per_rank,
    self.use_fp8_dispatch,
)

def dispatch(self, hidden_states, topk_output):
    # ... 前置校验省略 ...
    if not self.use_fp8_dispatch:
        # BF16 wire: 激活原样进入 ElasticBuffer, 不量化、不产生 scale;
        # 后续 DeepGEMM 预置换适配器也不再分配激活 scale 张量。
        dispatch_x = hidden_states
        use_tma_aligned_col_major_sf = False
    elif use_masked:
        # FP8 masked 路径: 保持原有 per-token-group 量化与 scale 打包。
        _ensure_fp8_quant_available()
        _ue8m0 = self.scale_format.ue8m0
        dispatch_x = sglang_per_token_group_quant_fp8(
            hidden_states, _SCALE_BLOCK_SIZE, ue8m0=_ue8m0
        )
    # ... 其余 FP8 分支 (TMA 对齐 scale 等) 省略 ...

```

评论区精华

该 PR 的 GitHub 页面没有留下公开 review 评论 (comments_count=0、review_comments_count=0)，但 commit 5a24eda 明确标注「[Fix] Address DeepEP v2 review feedback」，说明合入前存在一轮 review 迭代。从 commit 序列与 PR 描述可以还原四个核心设计决策：

- wire dtype 必须跟随 checkpoint 而不是自由配置：get_deepep_v2_dispatcher_output_dtype 在显式 flag 与 checkpoint 矛盾时直接抛错，PR 描述强调「Require the quantization method when creating the dispatcher so an omitted argument cannot silently select FP8」——这是典型的防静电错误设计。
- 确定性 reduce-scatter 用 all-reduce + 本地切片，而非改造原生 reduce-scatter：因为问题本质是求和顺序依赖接收 rank，而 all-reduce 让所有 rank 拿到同一归约结果；代价是通信量上升与额外临时 buffer，作者明确标注「End-to-end performance overhead has not been benchmarked」。
- ElasticBuffer receive sorting 保持关闭：专家计算行独立、combine 走 handle 的 slot 指针，sort 的动态索引与 decode CUDA graph 捕获不兼容，属于正确性优先于性能的取舍。
- 第 6 个 commit (5f190a7) 补上了 `reduce_scatterv`：commit message 指出 `should_use_dp_reduce_scatterv()` 在 DP attention (attn_dp_size>1) 时把 combine 路由到该路径，且 `dp_attention.py` 直接使用它——最初只改 `reduce_scatter_tensor` 会漏掉 DP 部署场景，这是 review 或自测中发现的关键缺口。
- 暂无高价值评论线程

风险与影响

- 风险：

1. 确定性路径的性能与通信开销：reduce_scatter_tensor 与 reduce_scatterv 在确定性模式下从原生 reduce-scatter 改为 all-reduce + 全输入临时 buffer + input.clone(), 通信量和显存临时占用上升，低延迟 decode 场景可能受影响；PR 描述明确未做端到端基准。
2. 确定性依赖底层 all-reduce 实现：4 卡回归固定 NCCL_ALGO=allreduce:tree 并关闭 custom all-reduce；如果生产环境走了自定义 all-reduce（如 FlashAttention custom AR）或其他 NCCL 算法，求和顺序是否仍稳定缺少运行时断言。
3. 混合量化 checkpoint 未覆盖：ElasticBuffer 单例缓存键包含 use_fp8_dispatch, BF16/FP8 层交替会触发层边界的集体缓冲区重建，PR 描述明确标注「mixed-quantization checkpoints need separate lifecycle coverage」。
4. BF16 dispatch 带宽成本：BF16 激活载荷约为 FP8 的两倍，ElasticBuffer 总分配不一定翻倍但通信压力上升。
5. CI 状态混杂：合并时 PR Test (Base) 通过，但 PR Test (Extra) 与 AMD ROCm 10 失败；对改动分布式通信层的 PR 而言，非主 CI 的失败需要后续确认是否与本次变更相关。

- 影响：

- 用户侧：--moe-a2a-backend deepep_v2 首次支持 BF16 checkpoint；
--enable-deterministic-inference 下批次增长不再改变已有 token 的生成结果（修复前 31.1% 的 BF16 元素因 rank 迁移而位级不一致）；FP8 用户的既有契约完整保留。
- 系统侧：GroupCoordinator.reduce_scatter_tensor 与 reduce_scatterv 的确定性分支属于分布式通信层全局路径，所有 TP/DP 模型在确定性开关下都会走新逻辑，不只影响 MoE；非确定性模式保持原生路径，零行为变化。
- 团队侧：新增 4 卡 B200 集体回归与 manual logprob 探针，CI 资源成本上升；PR 未附用户文档更新（checklist 中未勾选），后续需要补 --deepep-dispatcher-output-dtype 与确定性模式的行为说明。
- 验证范围：模型验证仅覆盖同构 BF16/FP8 的 Qwen3 系 checkpoint、单机 direct 模式；多节点 / 混合传输、其他模型族、以及 GSM8K 精度数据（修复前测得）均未覆盖。
- 风险标记：核心通信路径变更，确定性模式性能未量化，混合量化触发 buffer 重建，确定性依赖 all-reduce 底层实现，BF16 dispatch 带宽成本翻倍

关联脉络

- PR #35758 Add expert_start and num_experts to the contiguous scatter kernel: PR body 引用：给 _fwd_kernel_ep_scatter_2 增加专家范围参数，但未同步更新 DeepEP v2 调用者，是本 PR 修复的 TypeError 的直接来源。
- PR #35634 Introduce DeepEP v2 prefill permute via ep_scatter_from_psum: PR body 引用：引入 ep_scatter_from_psum 的调用者，缺参 bug 的载体；本 PR 为其补齐 expert_start 与 num_experts 实参。

- PR #38176 keeping router GEMM in fp32 for deterministic inference (DeepSeek V3/V4): 同一确定性推理主题: 该 PR 解决 MoE router GEMM 在确定性模式下的专家选择翻转, 本 PR 解决 TP reduce-scatter 的 rank 依赖求和, 两者共同构成 DeepEP 确定性推理正确性闭环。