PR #25855 完整报告

sgl-project/sglang

perf(jit_kernel/deepseek_v4): optimize paged_mqa_metadata

合并时间：2026-08-14 10:00

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/25855

执行摘要

- 一句话：DSv4 JIT 内核三路径重写，H200 最高加速 297×
- 推荐动作：值得精读。这是典型的"单 block 内核 → 多路径分派"性能工程案例：三路径分派的阈值选择与 static_assert 约束、Phase 3 串行扫描改并行二分的收益模型（O(bs) → O(log bs)）、与 deep_gemm 字节级对齐（逆序余数分配）的正确性前提、review 驱动的 workspace 所有权设计（CUDA Graph 安全），以及 benchmark 的 launch-bound 分析都很有借鉴价值。后续可关注作者提出的 multi-block scheduling follow-up。

功能与动机

PR body 明确指出性能瓶颈：旧内核 Phase 3 由每个 warp 的 lane 0 串行推进 q 遍历 per-batch 前缀数组，是 O(bs) 依赖型 smem 访问，整块逻辑只跑在 1 个 SM 上而其余 131 个 SM 空闲；bs ≥ 1024 时内核耗时 60–2000 μs。本次重写目标是把这一每步都可能执行的调度内核压到接近 launch floor，同时保持与生产基准 deep_gemm 的字节级一致性。

实现拆解

1. 内核重写（核心）：python/sglang/kernels/jit/csrc/deepseek_v4/paged_mqa_metadata.cuh 从 119 行扩到 386 行，引入 kTinyMax=64、kSmallMax=2048、kMBTileSize=4096 三个分派阈值与 kTinyBlock/kSmallBlock=256、kMBBlockSize=1024 的 block 配置；前缀和从手写两级 warp reduce 换成 cub::BlockScan<uint32_t, {256|1024}, BLOCK_SCAN_WARP_SCANS>，MetadataParams 删除 use_smem 开关。
2. Phase 3 并行化：旧实现是 lane 0 串行推进 q（O(bs) 依赖访问），新实现用 num_sm + 1 个目标对前缀数组做并行二分 upper_bound，每线程 O(log bs)（≤ 11 次依赖访问），并通过 for (i = tx; i <= num_sm; i += blockDim.x) 的 stride 写循环保证 num_sm > blockDim.x - 1 时输出表仍写满（保留上游 num_sm ≤ 1024 契约）。分配算法特意对齐 DeepGEMM 的"逆序余数"语义（前 pivot 个 SM 拿 avg、后 ret 个 SM 拿 avg + 1），这是字节级等价成立的前提。
3. Multi-block gmem 路径：bs > 2048 时启用 phase1_tile_scan_kernel（每 4096 个 batch 一个 block，写出 tile 内 inclusive prefix 和 tile sum），再由 schedule_from_tiles 单 block 汇总。per-block 静态 smem 恒定约 8 KB，因此能支撑 bs = 131072 而不触碰 sm_90 的 228 KB opt-in 上限——这正是 deep_gemm 在 bs ≥ 16384 直接失败、本实现仍保持约 6.4 μs 的原因。

4. Python 封装配套：python/sglang/kernels/ops/attention/dsv4/attn.py 的 get_paged_mqa_logits_metadata 在 bs > 2048 时用 torch.empty 分配 (bs + ceil(bs/4096)) 长度的 int32 workspace 传入 IndexerMetadataKernel::run，走 torch caching allocator，CUDA Graph 安全且生命周期由 Python 侧管理；这是 review 中"不要用 C++ 侧缓存 workspace"要求的落地。
5. 测试与基准配套：新增 test/registered/kernels/ops/attention/test_paged_mqa_metadata.py（5 组测试、双 oracle：生产基准 deep_gemm.get_paged_mqa_logits_metadata + 纯 PyTorch 算法规范 paged_mqa_metadata_ref，全部用 torch.equal 做字节级比对）；新增 benchmark/kernels/bench_paged_mqa_metadata.py，num_sm 运行时从 GPU 查询并迁移到 marker 框架；两者均登记 CI（base-b-kernel-unit 与 benchmark suite）。

关键文件：

- python/sglang/kernels/jit/csrc/deepseek_v4/paged_mqa_metadata.cuh（模块 JIT 内核；类别 source；类型 core-logic；符号 paged_mqa_metadata_tiny_kernel, paged_mqa_metadata_small_kernel, phase1_tile_scan_kernel, schedule_from_tiles）：内核本体重写（119 → 386 行），45.3× 加速全部来自这里：三路径分派、CUB BlockScan 前缀和、Phase 3 并行 upper_bound、multi-block gmem 路径。
- test/registered/kernels/ops/attention/test_paged_mqa_metadata.py（模块 内核测试；类别 test；类型 test-coverage；符号 paged_mqa_metadata_ref, _make_seq_lens, test_matches_pytorch_ref, test_matches_pytorch_ref_at_ksplitkv_boundary）：新增 5 组字节级 oracles 测试（deep_gemm 生产基准 + 纯 PyTorch 算法规范双基准），覆盖 kSplitKV=256 边界、multi-block 正确性下限、num_sm ∈ [1, 1024] 契约，是正确性信心的主要来源。
- python/sglang/kernels/ops/attention/dsv4/attn.py（模块 内核封装；类别 source；类型 entrypoint；符号 get_paged_mqa_logits_metadata）：Python 封装配合 workspace 所有权改造（review 驱动的关键设计决策）：bs > 2048 时由 Python 侧用 torch.empty 分配 scratch 并传入内核，保证 CUDA Graph 安全。
- benchmark/kernels/bench_paged_mqa_metadata.py（模块 内核基准；类别 source；类型 benchmark；符号 _make_seq_lens, benchmark）：新增 marker 框架 benchmark，num_sm 运行时从 GPU 查询（适配 H200/B200 等不同 SM 数），覆盖三条分派路径的形状包络并登记 CI。

关键符号：get_paged_mqa_logits_metadata, paged_mqa_metadata_tiny_kernel, paged_mqa_metadata_small_kernel, phase1_tile_scan_kernel, schedule_from_tiles, paged_mqa_metadata_ref, test_matches_pytorch_ref, test_matches_deep_gemm, test_byte_equal_at_correctness_floor, test_matches_pytorch_ref_at_large_num_sm

关键源码片段

python/sglang/kernels/jit/csrc/deepseek_v4/paged_mqa_metadata.cuh

内核本体重写（119 → 386 行），45.3× 加速全部来自这里：三路径分派、CUB BlockScan 前缀和、Phase 3 并行 upper_bound、multi-block gmem 路径。

```
// paged_mqa_metadata：按 batch_size 自适应三路径分派的核心改动。
```

```
// tiny(bs<=64) / small(bs<=2048) / multi-block(bs>2048)，取代旧版
// grid=1 单 block 内核里 lane 0 串行推进 q 的 O(bs) Phase 3。
constexpr uint32_t kTinyBlock = 256;
constexpr uint32_t kTinyMax = 64;
constexpr uint32_t kSmallBlock = 256;
constexpr uint32_t kSmallMax = 2048;
constexpr uint32_t kSmallItemsPerThread = 8; // 256 * 8 == 2048
static_assert(kSmallBlock * kSmallItemsPerThread == kSmallMax);

struct MetadataParams {
    uint32_t batch_size;
    uint32_t num_sm;
    const uint32_t* __restrict__ context_lens;
    uint32_t* __restrict__ schedule_metadata;
};

// bs <= 64：单 warp 0 做 inclusive scan，静态 smem 仅 256 B。
__global__ __launch_bounds__(kTinyBlock, 1)
void paged_mqa_metadata_tiny_kernel(const MetadataParams params) {
    __shared__ uint32_t s_prefix[kTinyMax];
    __shared__ uint32_t s_global_sum;

    const uint32_t tx = threadIdx.x;
    const uint32_t bs = params.batch_size;
    const uint32_t num_sm = params.num_sm;

    // Phase 1：把每个 batch 的工作量 ceil(len / 256) 扫成前缀和
    if (tx < 32) {
        uint32_t running = 0;
#pragma unroll
        for (uint32_t base = 0; base < kTinyMax; base += 32) {
            const uint32_t idx = base + tx;
            uint32_t v = 0;
            if (idx < bs) {
                const uint32_t length = params.context_lens[idx];
                v = (length + kSplitKV - 1) >> 8; // 256 = 2^8，等价于 ceil(length / 256)
            }
            // 单 warp 内 inclusive scan：每步用 __shfl_up_sync 级联前缀
#pragma unroll
            for (int o = 1; o < 32; o <<= 1) {
                uint32_t y = __shfl_up_sync(0xffffffff, v, o);
                if (tx >= static_cast<uint32_t>(o)) v += y;
            }
            v += running; // 接上上一段 32 元素的尾值
            if (idx < bs) s_prefix[idx] = v;
            running = __shfl_sync(0xffffffff, v, 31);
        }
        if (tx == 0) s_global_sum = running;
    }
```

```
    __syncthreads();

    const uint32_t global_sum = s_global_sum;
    const uint32_t avg = global_sum / num_sm;
    const uint32_t ret = global_sum % num_sm;
    const uint32_t pivot = num_sm - ret;

    // Phase 3 关键改动：不再由 lane 0 串行推进 q，而是让 num_sm + 1 个
    // 目标并行对前缀数组做二分 upper_bound，每线程 O(log bs)。
    // stride 循环保证 num_sm 超过 blockDim.x - 1 时也能完整写出全部行。
    for (uint32_t i = tx; i <= num_sm; i += blockDim.x) {
        // 对齐 DeepGEMM 的逆序余数分配：末尾 ret 个 SM 多分 1 份工作；
        // 总工作量小于 num_sm 时，靠前的空 SM 停在 (q=0, offset=0) 边界。
        const uint32_t target = i * avg + (i > pivot ? i - pivot : 0);

        uint32_t lo = 0, hi = bs;
        while (lo < hi) { // 二分找第一个前缀 > target 的位置，即 upper_bound
            const uint32_t mid = (lo + hi) >> 1;
            if (s_prefix[mid] <= target) lo = mid + 1;
            else hi = mid;
        }
        const uint32_t q = lo;

        if (q >= bs) {
            params.schedule_metadata[2 * i + 0] = bs;
            params.schedule_metadata[2 * i + 1] = 0;
        } else {
            const uint32_t prefix_prev = (q == 0) ? 0u : s_prefix[q - 1];
            params.schedule_metadata[2 * i + 0] = q;
            params.schedule_metadata[2 * i + 1] = target - prefix_prev;
        }
    }
}
```

test/registered/kernels/ops/attention/test_paged_mqa_metadata.py

新增 5 组字节级 oracles 测试（deep_gemm 生产基准 + 纯 PyTorch 算法规范双基准），覆盖 kSplitKV=256 边界、multi-block 正确性下限、num_sm ∈ [1, 1024] 契约，是正确性信心的主要来源。

```
# 纯 PyTorch 算法规范：int32 [num_sm + 1, 2] 划分表的 ground truth。
# 注意与 DeepGEMM 的逆序余数分配对齐：前 pivot 个 SM 拿 avg、
# 后 ret 个 SM 拿 avg + 1，保证总工作量小于 num_sm 时靠前的空 SM
# 停在 (q=0, offset=0) 边界而不是 q=batch_size。
def paged_mqa_metadata_ref(
    seq_lens: torch.Tensor, num_sm: int, page_size: int
) -> torch.Tensor:
    assert page_size == 64, f"page_size must be 64, got {page_size}"
    assert seq_lens.dtype == torch.int32, f"seq_lens dtype must be int32"
    assert seq_lens.dim() == 1, f"seq_lens must be 1-D, got {tuple(seq_lens.shape)}"
```

```
    device = seq_lens.device
    batch_size = int(seq_lens.shape[0])

    # 每个 batch 的工作量 = ceil(seq_len / kSplitKV)，kSplitKV = 256
    work_per_batch = (seq_lens.to(torch.int64) + KSPLITKV - 1) // KSPLITKV
    global_sum = int(work_per_batch.sum().item())
    avg = global_sum // num_sm
    ret = global_sum % num_sm
    pivot = num_sm - ret

    schedule_metadata = torch.empty((num_sm + 1, 2), dtype=torch.int32, device=device)
    work = work_per_batch.tolist()
    q = 0
    sum_work = work[0] if batch_size > 0 else 0
    for i in range(num_sm + 1):
        # 与 DeepGEMM 一致：前 pivot 个 SM 拿 avg，后 ret 个 SM 拿 avg + 1
        target = i * avg + max(i - pivot, 0)
        while sum_work <= target: # 串行推进 q（仅 oracle 用，内核已并行化）
            q += 1
            if q >= batch_size:
                break
            sum_work += work[q]
        if q >= batch_size:
            schedule_metadata[i, 0] = batch_size
            schedule_metadata[i, 1] = 0
        else:
            schedule_metadata[i, 0] = q
            schedule_metadata[i, 1] = target - (sum_work - work[q])
    return schedule_metadata


# 生产基准对标：与 deep_gemm 做字节等比；bs >= 16384 时 deep_gemm
# 超过 sm_90 smem 上限会自动跳过，但仍由上面的 PyTorch ref 覆盖。
# 断言用 torch.equal，无 atol/rtol——输出是确定性划分表，必须逐字节一致。
def test_matches_deep_gemm(bs: int, max_ctx: int):
    deep_gemm = _load_deep_gemm()
    seq_lens = _make_seq_lens(bs, max_ctx)
    got = get_paged_mqa_logits_metadata(seq_lens, PAGE_SIZE, NUM_SM)
    try:
        dg = deep_gemm.get_paged_mqa_logits_metadata(
            _to_2d_context_lens(seq_lens), PAGE_SIZE, NUM_SM
        )
    except RuntimeError as e:
        msg = str(e)
        if "smem" in msg.lower() or "capacity" in msg.lower():
            pytest.skip(f"deep_gemm smem cap exceeded at bs={bs}")
        raise
    assert torch.equal(got, dg)
```

python/sglang/kernels/ops/attention/dsv4/attn.py

Python 封装配合 workspace 所有权改造（review 驱动的关键设计决策）：bs > 2048 时由 Python 侧用 torch.empty 分配 scratch 并传入内核，保证 CUDA Graph 安全。

```
# JIT 内核的 Python 封装：workspace 由调用方（Python 侧）持有，
# 走 torch 的 caching allocator，保证 CUDA Graph 安全且生命周期可控——
# 这是 review 中“不要用 C++ 侧缓存 workspace”要求的落地。
def get_paged_mqa_logits_metadata(seq_lens: torch.Tensor, page_size: int, num_sm: int):
    assert page_size == 64
    seq_lens = seq_lens.view(-1).to(torch.int32)
    bs = int(seq_lens.shape[0])
    metadata = seq_lens.new_empty(num_sm + 1, 2)

    # 多 block 路径（bs > 2048）需要跨 kernel 的 scratch：tile 内前缀 + tile 和。
    # kMBTileSize 必须与 .cuh 中的常量保持一致（隐式契约，代码注释已标注）。
    if bs > 2048:
        kMBTileSize = 4096
        workspace = seq_lens.new_empty(
            bs + (bs + kMBTileSize - 1) // kMBTileSize, dtype=torch.int32
        )
    else:
        workspace = seq_lens.new_empty(0, dtype=torch.int32)

    module = _jit_metadata_module()
    module.run(seq_lens, metadata, workspace)
    return metadata
```

评论区精华

- “Ground truth 必须是 deep_gemm”：DarkSharpness 指出纯 PyTorch 实现不能作为基准——“The baseline must be deep_gemm's implementation (although their implementation is also very inefficient)”。作者按正确性与性能两个维度回应：新增 test_matches_deep_gemm 直接与生产基准字节比对，bs ≥ 16384 时 deep_gemm 超过 sm_90 smem 上限自动跳过，但仍有 PyTorch ref 覆盖。
- “不要用缓存的 workspace，很危险”：DarkSharpness 要求把辅助张量从 Python 侧传入。作者移除 cudaMalloc + unordered_map 缓存，改为 torch.empty 调用方持有，并解释未采用 torch.cumsum 链的原因（约 5 次 host launch、5–10 μs，比融合 kernel 的 3–4 μs 慢）。
- “能否用多个 block 做调度？”：作者给出 launch-bound 论证：num_sm + 1 = 133 个二分目标约 1 μs launch + 2 μs compute，拆多 block 会引入串行 launch 开销、在小 bs 下反而回归；提议作为 follow-up 提供带 profile 数据的对比。
- “基准要适配不同 GPU”：B200 有 148 个 SM，num_sm 硬编码 132 不合理；作者改为运行时查询 multi_processor_count，并验证 num_sm ∈ {132, 148, 257, 1024} 下三条路径字节等价。
- “rebase 并迁移到 marker 框架”：主干刚重构 benchmark 工具，作者把 benchmark 从 triton.testing.perf_report 迁到 @marker.parametrize / marker.do_bench，

DarkSharpness 最终评价 “The latency really LGTM. Nice work!”。

- 正确性基准必须是 deep_gemm 而非 Python 实现 (correctness): 作者新增 test_matches_deep_gemm 与 deep_gemm.get_paged_mqa_logits_metadata 做字节级比对；bs >= 16384 时 deep_gemm 超出 sm_90 smem 上限自动跳过，但仍由 PyTorch ref 覆盖。
- 禁止 C++ 侧缓存 workspace (design): 作者移除 cudaMalloc + unordered_map 缓存，改为 Python 侧 torch.empty 分配并经 IndexerMetadataKernel::run 传入；并说明未用 torch.cumsum 链的理由（约 5 次 host launch 5-10 μs，比融合 kernel 的 3-4 μs 慢）。
- Phase 3 能否用多 block 调度 (performance): 作者论证调度 kernel 是 launch-bound：num_sm+1=133 个二分目标约 1 μs launch + 2 μs compute，拆多 block 净收益最多 1 μs 且小 bs 会回归；提议作为 follow-up 提供带 profile 数据的对比，未在本 PR 改动。
- benchmark 的 num_sm 需适配不同 GPU (testing): 作者改为运行时查询 torch.cuda.get_device_properties(0).multi_processor_count，CUDA 不可用时回退 132；并验证 num_sm ∈ {132, 148, 257, 1024} 下三条路径与 PyTorch ref 字节等价。
- benchmark 迁移到新 marker 框架 (testing): 作者 rebase 主干并把 benchmark 从 triton.testing.perf_report 迁到 @marker.parametrize / @marker.benchmark / marker.do_bench；DarkSharpness 回复 “The latency really LGTM. Nice work!”。

风险与影响

- 风险：
  - Fallback 路径正确性风险：本 PR 优化的是 JIT fallback 内核，生产默认仍走 deep_gemm。若未来调用方切到 JIT 路径，依赖双 oracle 测试兜底；Phase 3 的逆序余数分配语义必须与 deep_gemm 严格一致，任何偏移都会被 test_matches_deep_gemm 捕获。
  - Workspace 隐式契约：attn.py 与 .cuh 之间依赖 kMBTileSize=4096 和 workspace 尺寸公式的隐式一致（代码注释已标注），后续任一文件改动都可能造成越界或错误结果；bs = 65536/131072 的 test_byte_equal_at_correctness_floor 覆盖了该路径。
  - 覆盖率缺口：作者明示 AMD / MUSA / SM100+ 未测试；虽无 SM 特定 intrinsic 且静态 smem ≤ 8 KB，但 CUB 版本差异与编译器行为在非 NVIDIA 平台仍是未知数。
  - 依赖变化：sgl_kernel/warp.cuh 依赖替换为 cub/block/block_scan.cuh，引入对 CUB 的新耦合；旧手写 reduce 行为不再被验证。
  - 长周期合并风险：PR 含 32 个 commit，期间经历 #25884 对 deepseek_v4.py 的拆分重构，冲突合并正确性已由最终 e2e 测试（fp4/fp8 B200/H200 + CP）覆盖；CI 还出现过 GitHub API rate limit 导致的假失败。
- 影响：
  - 性能影响：DSv4 大 batch decode/prefill 的调度元数据生成从 60–2000 μs 降到约 6.5 μs 封顶，对长上下文、大并发场景的每步延迟有明显改善；bs ≤ 64 的小 batch 也有 1.5–2.5× 收益，bs = 8192 相对 deep_gemm 快 5.02×。

- 用户影响：公共 API IndexerMetadataKernel::run 与 get_paged_mqa_logits_metadata 签名不变，nsa_backend、nsa/nsa_indexer、dsv4/metadata 等 8 个调用方零改动，对用户透明。
- 团队影响：建立了“JIT 内核字节级双 oracle 验证 + marker benchmark + CI 注册”的完整配套范式，对后续 DSv4 / deep_gemm 对标工作有示范价值。
- 范围限制：默认生产路径（deep_gemm）未改动，收益主要体现在 JIT fallback 场景。
- 风险标记：核心内核重写，JIT fallback 路径，workspace 隐式契约，AMD/MUSA/SM100 未验证，新增 CUB 依赖

关联脉络

- PR #23882 Introduce paged_mqa_metadata JIT kernel: PR body 明确点名本 PR 优化的 paged_mqa_metadata.cuh 由 #23882 引入，本 PR 是其性能续作。
- PR #25884 JIT cleanup: split deepseek_v4.py into dsv4/*.py: 合并主干的 commit 中显式处理了与 #25884 的冲突：接受上游对 jit_kernel/deepseek_v4.py 的删除，并把 workspace 分配移植进 dsv4/attn.py，最终调用链落在 sglang/kernels/ops/attention/dsv4/。
- PR #25274 Benchmark marker framework: 本 PR 最终版的 benchmark 迁移到 #25274 引入的 sglang.jit_kernel.benchmark.marker API（commit 4453865 中注明）。
- PR #33857 [Perf] Skip trivial DSV4 nonpaged indexer logits: 同属 DSV4 索引器 / 元数据性能维护线，说明 DSV4 是当前持续深挖性能的重点模型。