PR #34257 完整报告

sgl-project/sglang

[JIT Kernel] Migrate per-token FP8 quantization from AOT to JIT

合并时间：2026-08-11 20:40

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/34257

执行摘要

- 一句话：per-token FP8 量化迁移至 JIT，兼容 AOT 并提速
- 推荐动作：值得精读。重点关注三处设计：一是按架构选择 math mode（SM90 fast_math、SM100+ precise_math）以保证位精确的思路；二是 CTA/warp 双 dispatch 阈值与零行 legacy 行为的刻意保留及对应测试设计；三是内核注册表从 AOT 切到 JIT 的接入模式（可作为后续 AOT→JIT 迁移的标准操作流程）。测试文件 test/registered/kernels/ops/quantization/test_per_token_quant_fp8.py 是位精确迁移测试的样板。

功能与动机

PR body 明确指出："Migrate CUDA per-token FP8 activation quantization from the prebuilt sgl_kernel path to SGLang's lightweight JIT kernel infrastructure. This reduces the CUDA runtime's dependency on the AOT wheel for this operation while preserving the existing output-buffer contract, numerical behavior, and public compatibility API." 即核心动机是降低 CUDA serving 对 AOT wheel 的依赖，同时把正确性保障提升到 " 位精确 " 级别。

实现拆解

1. JIT CUDA 内核移植：新增 python/sglang/kernels/jit/csrc/gemm/per_token_quant_fp8.cuh，实现 warp 核（每 warp 处理一个 token，warp::reduce_max 归约）与 CTA 核（每 CTA 一个 token，共享内存归约）两条 dispatch；launch_per_token_quant_fp8 按 num_tokens >= sm_count * 2 * 8 选择 warp 路径，与 AOT 阈值保持一致。host 侧通过 TensorMatcher 与 CHECK_HOST 强制 hidden_dim % 4 == 0、output 行数 >= 输入行数（padded 契约）等约束。
2. Python 侧 JIT 接入：新增 python/sglang/kernels/ops/quantization/per_token_quant_fp8.py，用 cache_once 按 dtype 缓存编译产物；按 arch.major == 9（SM90）启用 fast_math 以匹配 AOT 在 H100/H200 上的编译行为，SM100+ 走 precise_math；通过 register_custom_op 声明 output_q/output_s 为 in-place 修改参数，提供 torch.compile 兼容的元实现。
3. 路由与兼容性调整：python/sglang/kernels/ops/quantization/__init__.py 将 quantization.sgl_per_token_quant_fp8 的注册后端从 KernelBackend.AOT 切换为 KernelBackend.JIT；fp8_kernel.py 中 CUDA 分支改走新 JIT 包装、MUSA 分支保留 sgl_kernel AOT 导入，并删除旧的 register_fake_if_exists fake op；

common_extension.cc 保留 sgl_per_token_quant_fp8 的 torch 导出作为外部消费者兼容 API。

4. 测试与基准迁移：删除 python/sglang/kernels/aot/tests/test_per_token_quant_fp8.py 与 aot/benchmark/bench_per_token_quant_fp8.py，在 test/registered/kernels/ 下新增位精确差分测试（覆盖 FP16/BF16/FP32、CTA/warp dispatch、非 2 的幂 hidden_dim、零行、FP8 中点四舍五入、padded 输出尾部保持）与 JIT-vs-AOT 注册基准；B200 通过 base-b-kernel-unit-test-4-gpu-b200 验证 SM100 redux.sync.max.f32 归约路径。

关键文件：

- python/sglang/kernels/jit/csrc/gemm/per_token_quant_fp8.cuh（模块 量化内核；类别 source；类型 core-logic；符号 per_token_quant_fp8_warp_kernel, per_token_quant_fp8_cta_kernel, launch_per_token_quant_fp8, per_token_quant_fp8）：JIT CUDA 内核的核心实现，包含 warp/CTA 双 dispatch 量化内核、launch 阈值决策与 host 侧 padded 契约校验，是整个迁移的基石。
- python/sglang/kernels/ops/quantization/per_token_quant_fp8.py（模块 JIT 接入；类别 source；类型 infrastructure；符号 _jit_per_token_quant_fp8_module, per_token_quant_fp8）：JIT 模块加载与 custom-op 包装的接入点，包含按架构选择 math mode 的关键位精确策略与 torch.compile 兼容注册。
- test/registered/kernels/ops/quantization/test_per_token_quant_fp8.py（模块 量化测试；类别 test；类型 test-coverage；符号 _run_impl, _assert_bitwise_equal, _warp_dispatch_num_tokens, test_per_token_quant_fp8_is_bit_exact）：位精确差分测试的核心文件，覆盖 dtype、双 dispatch、零行、FP8 中点舍入与 padded 输出契约，是本次迁移正确性的主要保障。
- python/sglang/kernels/ops/quantization/__init__.py（模块 内核注册；类别 infra；类型 infrastructure）：内核注册表将 per-token FP8 量化从 AOT 后端切换为 JIT 后端，是 serving 路由切换的关键点。
- python/sglang/kernels/ops/quantization/fp8_kernel.py（模块 量化入口；类别 source；类型 infrastructure；符号 _）：serving 侧入口 scaled_fp8_quant 的分支调整，CUDA 改走 JIT、MUSA 保留 AOT，并移除旧的 fake op 注册。
- python/sglang/kernels/aot/csrc/common_extension.cc（模块 兼容接口；类别 source；类型 core-logic）：保留 sgl_per_token_quant_fp8 的 torch 导出，作为外部 sgl_kernel 消费者的兼容 API 与差分测试的 AOT 参照。
- python/sglang/kernels/aot/benchmark/bench_per_token_quant_fp8.py（模块 基准测试；类别 test；类型 deletion；符号 torch_per_token_quant_fp8, vllm_per_token_quant_fp8, sglang_per_token_quant_fp8, calculate_diff）：删除旧 AOT benchmark（含 torch/vLLM/SGLang 三路对比与 triton perf_report），迁移到 registered 布局下的新基准。
- python/sglang/kernels/aot/tests/test_per_token_quant_fp8.py（模块 量化测试；类别 test；类型 deletion；符号 torch_per_token_quant_fp8, sglang_per_token_quant_fp8, test_per_token_quant_compare_implementations）：删除旧的 AOT 对照测试，其覆盖由新的位精确差分测试取代（从容差比较升级为逐位比较）。

- test/registered/kernels/benchmark/quantization/bench_per_token_quant_fp8.py（模块基准测试；类别 test；类型 test-coverage；符号 _jit_quant, benchmark）：新注册的 JIT-vs-AOT 基准，沿用 marker 框架，覆盖 FP16/BF16、多种 token 数与 hidden_dim。

关键符号：per_token_quant_fp8, _jit_per_token_quant_fp8_module, per_token_quant_fp8_warp_kernel, per_token_quant_fp8_cta_kernel, launch_per_token_quant_fp8, scaled_fp8_quant, sgl_per_token_quant_fp8

关键源码片段

python/sglang/kernels/jit/csrc/gemm/per_token_quant_fp8.cuh

JIT CUDA 内核的核心实现，包含 warp/CTA 双 dispatch 量化内核、launch 阈值决策与 host 侧 padded 契约校验，是整个迁移的基石。

```
// warp 核：每个 warp 负责一行 token，先按 lane 分块求 abs max 并跨 lane
// reduce，得到该行 scale = max / FP8_E4M3_MAX；scale 为 0（整行全零）时
// scale_inv 置 0 而非除零，这是 AOT warp 路径的 legacy 行为，必须原样保留
template <typename T, int kVecSize>
__global__ void per_token_quant_fp8_warp_kernel(
    const T* __restrict__ input,
    fp8_e4m3_t* __restrict__ output_q,
    float* __restrict__ output_s,
    uint32_t hidden_dim,
    uint32_t num_tokens) {
  using namespace device;
  using input_vec_t = AlignedVector<T, kVecSize>;
  using output_vec_t = AlignedVector<fp8_e4m3_t, kVecSize>;

  const uint32_t warp_id = threadIdx.x / kPerTokenQuantWarpSize;
  const uint32_t lane_id = threadIdx.x % kPerTokenQuantWarpSize;
  const uint32_t token_id = blockIdx.x * kPerTokenQuantTokensPerCTA + warp_id;
  if (token_id >= num_tokens) {
    return;
  }

  const T* token_input = input + token_id * hidden_dim;
  fp8_e4m3_t* token_output = output_q + token_id * hidden_dim;
  const uint32_t num_vecs = hidden_dim / kVecSize;

  float max_value = 0.0f;
  for (uint32_t i = lane_id; i < num_vecs; i += kPerTokenQuantWarpSize) {
    input_vec_t input_vec;
    input_vec.load(token_input, i);
#pragma unroll
    for (int j = 0; j < kVecSize; ++j) {
      max_value = math::max(max_value, math::abs(static_cast<float>(input_vec[j])));
    }
  }
```

```
  const float scale = warp::reduce_max(max_value) / math::FP8_E4M3_MAX;
  if (lane_id == 0) {
    output_s[token_id] = scale;
  }
  const float scale_inv = scale == 0.0f ? 0.0f : 1.0f / scale;

  // 量化主循环：乘以 scale_inv 后 clamp 到 [-FP8_E4M3_MAX, FP8_E4M3_MAX]，
  // 再窄化到 fp8_e4m3_t，逐元素顺序与 AOT 完全一致，避免舍入漂移
  for (uint32_t i = lane_id; i < num_vecs; i += kPerTokenQuantWarpSize) {
    input_vec_t input_vec;
    output_vec_t output_vec;
    input_vec.load(token_input, i);
#pragma unroll
    for (int j = 0; j < kVecSize; ++j) {
      const float value = static_cast<float>(input_vec[j]) * scale_inv;
      output_vec[j] = static_cast<fp8_e4m3_t>(math::max(math::min(value, math::FP8_E4M3_
      MAX), -math::FP8_E4M3_MAX));
    }
    output_vec.store(token_output, i);
  }
}

// 双 dispatch 阈值与 AOT 对齐：token 数 >= sm_count * 16 时启用 warp 核，
// 否则退化到每 CTA 一行（CTA 核没有 scale==0 保护，同样属于保留行为）
template <typename T, int kVecSize>
void launch_per_token_quant_fp8(
    DLDevice device, const T* input, fp8_e4m3_t* output_q, float* output_s,
    uint32_t hidden_dim, uint32_t num_tokens) {
  constexpr uint32_t kBlockSize = 256;
  const uint32_t sm_count = host::runtime::get_sm_count(device.device_id);
  const bool use_warp_kernel = num_tokens >= sm_count * 2 * kPerTokenQuantTokensPerCTA;
  if (use_warp_kernel) {
    const uint32_t grid = host::div_ceil(num_tokens, kPerTokenQuantTokensPerCTA);
    host::LaunchKernel(grid, kBlockSize, device)(
        per_token_quant_fp8_warp_kernel<T, kVecSize>, input, output_q, output_s, hidden_dim,
        num_tokens);
  } else {
    host::LaunchKernel(num_tokens, kBlockSize, device)(
        per_token_quant_fp8_cta_kernel<T, kVecSize>, input, output_q, output_s, hidden_dim);
  }
}

// host 侧契约：输出缓冲允许比输入多行（serving padding），内核只写
// 前 num_tokens 行，padding 尾部保持原值；hidden_dim 必须能被 4 整除
CHECK_HOST(MOutput.unwrap() >= M.unwrap())
    << "per_token_quant_fp8: output buffers must have at least "
    << M.unwrap() << " rows, got " << MOutput.unwrap();
```

python/sglang/kernels/ops/quantization/per_token_quant_fp8.py

JIT 模块加载与 custom-op 包装的接入点，包含按架构选择 math mode 的关键位精确策略与 torch.compile 兼容注册。

```
# 按 dtype 缓存 JIT 编译产物，避免每次调用重复走编译流程
@cache_once
def _jit_per_token_quant_fp8_module(dtype: torch.dtype) -> Module:
    if dtype not in (torch.float16, torch.bfloat16, torch.float32):
        raise RuntimeError(
            f"Unsupported dtype {dtype}. Supported: float16, bfloat16, float32"
        )
    arch = get_jit_cuda_arch()
    # SM90 上 AOT wheel 以 fast math 编译，JIT 必须复刻同一 math mode
    # 才能在 H100/H200 上做到逐位一致；SM100+ 则走 precise_math
    use_fast_math = (arch.major, arch.minor) == (9, 0)
    math_mode = "fast_math" if use_fast_math else "precise_math"
    args = make_cpp_args(dtype)
    return load_jit(
        "per_token_quant_fp8",
        math_mode,
        *args,
        cuda_files=["gemm/per_token_quant_fp8.cuh"],
        cuda_wrappers=[("per_token_quant_fp8", f"per_token_quant_fp8<{args}>")],
        extra_cuda_cflags=["--use_fast_math"] if use_fast_math else [],
    )


# register_custom_op 提供 torch.compile 兼容的 meta 实现；
# output_q / output_s 声明为 in-place 修改参数
@register_custom_op(
    op_name="per_token_quant_fp8",
    mutates_args=["output_q", "output_s"],
)
def per_token_quant_fp8(
    input: torch.Tensor,
    output_q: torch.Tensor,
    output_s: torch.Tensor,
) -> None:
    """Dynamically quantize each row to FP8 E4M3."""
    module = _jit_per_token_quant_fp8_module(input.dtype)
    # scale 以 (rows, 1) 形状传给内核，与 AOT 的输出契约一致
    module.per_token_quant_fp8(input, output_q, output_s.view(output_s.shape[0], 1))
```

test/registered/kernels/ops/quantization/test_per_token_quant_fp8.py

位精确差分测试的核心文件，覆盖 dtype、双 dispatch、零行、FP8 中点舍入与 padded 输出契约，是本次迁移正确性的主要保障。

```
# 位精确断言：直接比较 uint8 视图，保证 JIT 与 AOT 输出 /scale 逐位一致
def _assert_bitwise_equal(actual: torch.Tensor, expected: torch.Tensor):
    assert torch.equal(actual.view(torch.uint8), expected.view(torch.uint8))
```

```
# warp 与 CTA 两条 dispatch 都必须覆盖：通过行数越过 / 低于
# sm_count * 16 的阈值，确保两条 kernel 路径都被差分验证
def _warp_dispatch_num_tokens() -> int:
    return torch.cuda.get_device_properties(0).multi_processor_count * 16


@pytest.mark.parametrize("dtype", [torch.float16, torch.bfloat16, torch.float32])
@pytest.mark.parametrize("dispatch", ["cta", "warp"])
@pytest.mark.parametrize("hidden_dim", [1076, 1368])
def test_per_token_quant_fp8_is_bit_exact(dtype, dispatch, hidden_dim):
    """JIT 迁移必须保留 AOT 的每一个输出与 scale 位。"""
    num_tokens = 39 if dispatch == "cta" else _warp_dispatch_num_tokens()
    input = torch.rand((num_tokens, hidden_dim), dtype=dtype, device="cuda")

    actual_output, actual_scale = _run_impl(input, use_jit=True)
    expected_output, expected_scale = _run_impl(input, use_jit=False)

    _assert_bitwise_equal(actual_scale, expected_scale)
    _assert_bitwise_equal(actual_output, expected_output)
```

评论区精华

该 PR 无 review 评论，BBuf 直接 APPROVED。正确性论证主要由 PR body 的 Accuracy Tests 与 Speed Tests 小节承担：H200（SM90）与 B300（SM103）各 32 个测试通过，覆盖位精确 AOT-vs-JIT 差分；基准矩阵覆盖 FP16/BF16、token 数 1..7807、hidden_dim 512..4096。

- 暂无高价值评论线程

风险与影响

- 风险：
  - 位精确强约束：SM90 依赖 fast_math 编译对齐 AOT，若 AOT wheel 后续改变编译参数或 sgl_kernel 升级实现，差分测试会先暴露漂移，但已合入后若跳过测试则 serving 输出可能静默变化。
  - legacy 行为保留：CTA 核零行路径 1.0f / scale_smem 会产生 inf/NaN（无 scale == 0 保护），这是刻意保留的 AOT 行为；依赖 IEEE 语义，在非 NVIDIA 平台未验证。
  - MUSA 仍依赖 AOT：fp8_kernel.py 中 MUSA 分支继续走 sgl_kernel，后续需单独迁移，否则 AOT wheel 裁剪会破坏 MUSA 路径。该分支无新增测试。
  - hidden_dim 约束收紧：JIT 校验 hidden_dim % 4 == 0 并显式报错，若曾有 consumer 传入非 4 倍数维度且未触发 AOT 报错，迁移后可能提前失败。
  - 观测性损失：删除的 AOT benchmark 含 vLLM 横向对比，迁移后失去与 vLLM 的对照数据。
  - 影响：影响所有 CUDA 上 FP8 动态 per-token 量化路径（scaled_fp8_quant 的 use_per_token_if_dynamic=True，涉及 DeepSeek 等 FP8 模型 serving）；通过 sglang.kernels.ops.quantization.per_token_quant_fp8 与内核注册表接入，外部

sgl_kernel.sgl_per_token_quant_fp8 消费者不受影响。性能收益显著（大 token 数下最高 2-3x，几何平均约 1.06-1.10x）。团队层面，该 PR 提供了 "AOT 算子迁移到 JIT+ 位精确差分测试 " 的完整范例，可复用为后续迁移模板。

- 风险标记：位精确依赖 math mode 特判，核心 FP8 serving 路径变更，MUSA 仍依赖 AOT wheel，删除 vLLM 横向对比基准，hidden_dim 对齐约束收紧

关联脉络

- PR #34329 HiSparse: shared-index (IndexShare) plan-then-IO swap-in prefetch: 同为基于 SGLang JIT kernel 基础设施新增内核的 PR，代表 AOT wheel 向轻量 JIT 迁移的同一技术方向。
- PR #34305 [diffusion] weight-only FP8: dequantize linear weights once at first use (Ideogram-4 denoise -18.8% H200 / -7.8% H100, bit-exact): 同为 FP8 量化性能优化 + bit-exact 验证路线，共用 quant/jit-kernel 技术栈。
- PR #34306 [diffusion] ERNIE-Image: fuse rotate-half RoPE + GELU-mul and hoist rope cos/sin (denoise -16.2% H100 / -12.7% H200, bit-exact): 同为 jit-kernel + performance + bit-exact 方向，体现仓库正在系统性用 JIT 内核替代 / 优化预编译算子。
- PR #34136 [Diffusion] Add online FP8 support for Krea-2: 同为 FP8 量化能力线演进，扩展动态量化（scaled_fp8_quant 生态）的应用面。