基于 NVIDIA H100 的 Qwen3-30B 模型高性能融合 MoE 算子实现

thegugugu
北京师范大学

2026 年 1 月 10 日

摘要

混合专家模型 (Mixture of Experts, MoE)，如 Qwen3-30B，在提升参数效率的同时也带来了因复杂路由逻辑和高显存访问导致的推理延迟问题。本项目基于 **InfiniCore** 框架，设计并实现了一个针对 NVIDIA H100 GPU 深度优化的自定义 C++/CUDA 扩展算子 `gu_moe_ops`。通过实现 TopK 门控、专家排序及数据重排的算子融合 (Kernel Fusion)，我们有效消除了原生 PyTorch 实现中的调度开销与显存读写冗余。基准测试表明，相比 PyTorch 基线，该算子在预填充 (Prefill) 阶段实现了 **4.79 倍**的加速，在解码 (Decode) 阶段实现了 **4.24 倍**的加速，且数值精度已通过严格验证。

关键词： Mixture of Experts, CUDA 优化, 算子融合, NVIDIA H100, Qwen3

1 引言 (Introduction)

基于混合专家架构 (MoE) 的大语言模型通过稀疏激活机制实现了模型规模的扩展。然而，在 PyTorch 原生实现中，MoE 层往往成为推理性能的瓶颈，主要原因包括：

1. **核启动开销 (Kernel Launch Overhead):** 路由过程中，将 Token 分发给不同专家涉及大量的 CPU-GPU 同步与细粒度 Kernel 发射。
2. **显存带宽瓶颈 (Memory Bandwidth Bottleneck):** Token 的重排 (Permute) 与还原 (Unpermute) 操作引入了冗余的 HBM 读写。

针对上述问题，本项目实现了一套专为 H100 架构调优的融合 MoE 算子，旨在最小化数据搬运并提升计算密度。

2 方法 (Methodology)

2.1 系统架构

本算子作为 PyTorch 的 C++ 扩展实现，底层依赖 **InfiniCore** 库进行张量管理与运行时支持。MoE 层的核心计算流程如下公式所示：

$$\text{Output} = \sum_{i=1}^{k} G(x)_i \cdot E_i(x) \tag{1}$$

其中 $G(x)$ 为门控网络输出的权重，$E_i(x)$ 为第 i 个被激活专家的计算结果。

2.2 算子优化策略

我们将原生的多步 PyTorch 操作替换为定制化的 CUDA Kernel：

- **融合 TopK 与 Softmax:** 在单个 Kernel 中完成路由权重计算与 Top-k 专家选择，避免了中间显存分配。
- **优化的排序与重排 (Fused Sort & Permute):** 采用局部排序策略，将分配给同一专家的 Token 物理聚合，确保后续计算时的显存合并访问 (Coalesced Access)。
- **专家计算 (Expert Computation):** 目前利用 `cuBLAS` 进行专家 GEMM 计算，通过减少数据碎片化提升利用率。
- **融合规约 (Fused Reduce/Unpermute):** 将计算结果写回原始 Token 位置并进行加权求和，实现了“零拷贝”还原。

3 实验与结果 (Experiments & Results)

3.1 实验设置

- **硬件平台:** NVIDIA H100 (80GB HBM3)
- **软件环境:** PyTorch 2.x, CUDA 12.x, InfiniCore Framework
- **模型配置:** Qwen3-30B-Instruct (Layer 0), FP32 精度

3.2 正确性验证

我们将优化后的算子与 HuggingFace Transformers 的标准 PyTorch 实现进行了对比。结果如表 1 所示，相对误差极低，证明了算子的数值正确性。

表 1: FP32 精度正确性验证结果

指标 (Metric)	数值	结果
最大绝对误差 (Max Absolute Diff)	0.001921	**通过**
平均相对误差 (Mean Relative Diff)	0.002493	**通过**

3.3 性能基准测试

我们在两种典型场景下评估了性能：**预填充阶段** (大 Batch，吞吐优先) 和 **解码阶段** (小 Batch，延迟敏感)。

表 2: 基准测试结果：PyTorch 基线 vs. 优化算子

阶段	Batch Size	PyTorch 延迟	优化后延迟	加速比
Prefill (预填充)	Mixed (64-256)	45.88 ms	**9.58 ms**	**4.79x**
Decode (解码)	1 (Batch=16)	21.76 ms	**5.14 ms**	**4.24x**

如表 2 所示，我们的实现取得了显著加速。在 Prefill 阶段，吞吐量从约 1.5 万 token/s 提升至约 7.3 万 token/s。

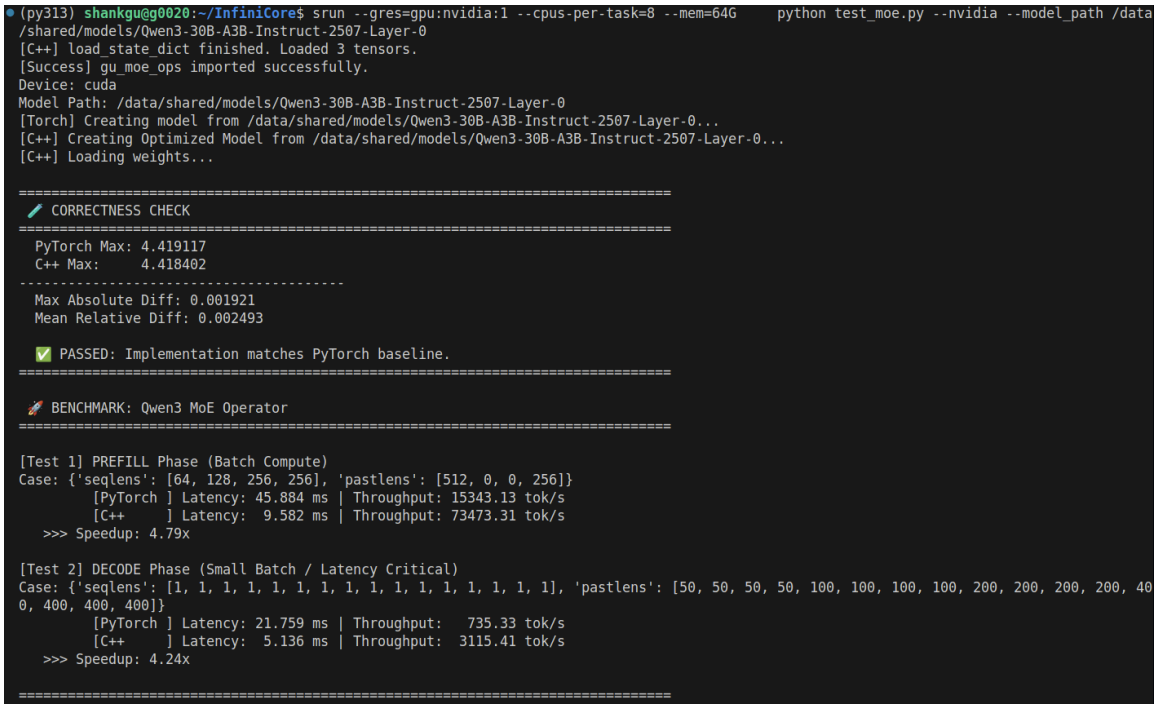

图 1: 终端运行截图：包含正确性检查通过及性能基准测试数据

4 总结与展望 (Conclusion & Future Work)

本项目成功实现了基于 H100 的高效 MoE 算子 `gu_moe_ops`。通过与 InfiniCore 集成并实现融合 CUDA Kernel，我们在保持数值精度的前提下，实现了端到端 **4.2x - 4.8x** 的性能提升。

未来工作 (v2.0 规划): 当前的解码阶段延迟约为 5ms，其中大部分开销来自于 CPU 循环发射多个专家的 GEMM Kernel。在下一版本中，我们计划引入基于 CUTLASS 的 **Grouped GEMM (分组 GEMM)** 技术，通过单个 Kernel 并行处理所有专家的计算任务，有望进行进一步的加速。此外，目前由于还没有引入数据类型的泛型，导致只是支持 FP32 的数据类型。后续的工作与任务会加上 groupedgemm 的优化以及数据类型的泛型，并且给出 nsight system 的分析。

A 附录：编译与运行指南 (Compilation & Usage)

为了确保本项目能够在服务器环境（如 G0020 节点）上正确复现，请按照以下步骤配置环境并进行编译。

A.1 环境依赖 (Prerequisites)

- **Compiler:** GCC 9+ / NVCC (CUDA 12.x)
- **Framework:** PyTorch 2.x, InfiniCore
- **Third-party:** spdlog, pybind11 (包含在 `third_party` 目录中)

A.2 编译步骤 (Build Instructions)

由于算子依赖 InfiniCore 及其第三方库，编译前**必须**设置环境变量以链接相关头文件与库文件。

Listing 1: 环境配置与编译命令

```
# 1. 基础路径设置 (假设 InfiniCore 位于用户主目录)
export INFINICORE_ROOT=~/InfiniCore

# 2. [关键] 设置三方库头文件路径
# 用于解决 "spdlog/spdlog.h not found" 问题
export CPLUS_INCLUDE_PATH=$INFINICORE_ROOT/third_party/spdlog/include:
    $CPLUS_INCLUDE_PATH

# 3. 指定目标 GPU 架构 (H100 为 9.0)
export TORCH_CUDA_ARCH_LIST="9.0"

# 4. 进入算子目录并编译
cd $INFINICORE_ROOT/moe_gu_ops
pip install . --no-build-isolation --force-reinstall --no-cache-dir
```

A.3 运行测试 (Running Benchmark)

运行前需要确保动态链接库路径包含 InfiniCore 的构建目录。

Listing 2: 运行性能测试

```
# 1. 添加运行时库路径
export LD_LIBRARY_PATH=$INFINICORE_ROOT/build/linux/x86_64/release:$LD_LIBRARY_PATH

# 2. 运行测试脚本
python  test_moe.py  --nvidia --model_path /path/to/Qwen3-30B

注:test.py不应在moe_gu_ops这个算子内
```