赛题：基于 Quest 算法的 GQA Sparse Attention 优化实现

1. 赛题简介

在大型语言模型（LLM）的解码阶段，标准的注意力机制（Full Attention）需要对当前生成的 token 与历史序列中所有 token 进行计算。这导致计算复杂度与序列长度呈**二次增长**，成为长序列生成时的主要瓶颈。

Sparse Attention 旨在通过**智能选择**历史序列中对当前 token 最重要的部分 KV 对，近似计算注意力，从而大幅降低计算开销。本赛题要求实现一个基于 **Quest 算法**、支持 **GQA（Grouped-Query Attention）** 的 Sparse Attention 内核，以达到性能显著优于参考 PyTorch 实现的目标。

核心思想分为两步：

1. **Quest 阶段**：利用 KV 缓存中每个“页面”（包含多个 token）的统计信息（如 min/max），快速为每个 KV 头（K Head）筛选出对当前 Query 最重要的 `topk` 个页面。
2. **GQA 阶段**：仅对 Quest 阶段选出的 KV 对进行 GQA 注意力计算，得到最终输出。

下面给出 Quest 算法和 GQA 的链接，请同学们自行学习
Quest 算法
GQA 介绍

2. 问题定义与输入输出

目标函数：实现高效的 `sparse_attn_triton` 函数，其功能与提供的参考实现 `sparse_attn_ref` 一致，但性能更高。

2.1 函数

```
def sparse_attn_triton(
    Q,                  # [batch_size, 1, num_q_heads, d]
    k_cache,            # [batch_size, max_seq, num_kv_heads, d]
    v_cache,            # [batch_size, max_seq, num_kv_heads, d]
    Q_HEAD_NUM,         # num_q_heads
    K_HEAD_NUM,         # num_kv_heads
    BLOCK_PAGE,         # 每个 kv page 包含的 token 数量
    topk,               # 每个 K 头选择的 top-k token 数量，方便起见设成 BLOCK_PAGE 的倍数
):
    B, _, _, d = Q.shape
    _, max_seq, _, _ = k_cache.shape

    GROUP_HEAD_NUM = Q_HEAD_NUM // K_HEAD_NUM
    PAGE_NUM = max_seq // BLOCK_PAGE
    topk_pages = topk // BLOCK_PAGE

    sm_scale = 1.0 / math.sqrt(d)
    attn_out = torch.empty_like(Q)

    # todo
    return attn_out
```

`batch_size (B)`: 批次大小，即同时处理多少个独立的序列。
`num_q_heads (H_q)`: Query 头的数量。在多头注意力中，每个头可以关注不同的模式。
`num_kv_heads (H_kv)`: Key 和 Value 头的数量。在 GQA 中，多个 Query 头共享一组 Key 和 Value 头，因此
`max_seq`: 表示 KV 缓存的最大容量
`d`: 每个注意力头的特征维度。

注：你可以自己设计添加多个 kernel，并通过 `sparse_attn_triton` 调用这些 kernel 端到端计算出最终的 sparse attention。

2.2 输入张量规格

- `Q` **(Query)**:
  - 形状： `[batch_size, 1, num_q_heads, d]`
  - 含义：当前解码步的 Query 向量。 `1` 表示每次生成一个 token。
- `k_cache` , `v_cache` **(KV 缓存)**:
  - 形状： `[batch_size, max_seq, num_kv_heads, d]`

    - 含义：存储了自生成开始以来所有历史 token 的 Key (`k_cache`) 和 Value (`v_cache`) 向量。
- **参数**:
    - `Q_HEAD_NUM` : 查询头总数，记为 `H_q` 。
    - `K_HEAD_NUM` : 键值头总数，记为 `H_kv` 。通常 `H_kv < H_q` 以实现 GQA。
    - `BLOCK_PAGE` : 定义“页面”的大小，即多少个连续的 token 被视作一个整体进行初筛。例如 64。
    - `topk` : 每个 KV 头经过初筛后，最终参与注意力计算的 token 数量。必须为 `BLOCK_PAGE` 的整数倍。
- **约束**:
    - `num_q_heads` 必须是 `num_kv_heads` 的整数倍。即 `GROUP_HEAD_NUM = Q_HEAD_NUM // K_HEAD_NUM` ，表示每个 KV 头对应一组 Q 头。

2.3 输出张量规格

- **返回值**: 一个形状为 `[batch_size, 1, num_q_heads, d]` 的 `torch.Tensor` 。
- **含义**: 经过 Sparse GQA 注意力计算后，当前步的注意力输出。应**数值等价**于参考实现 `sparse_attn_ref` 的输出。

3. 实现目标与评分核心

1. **正确性**: 实现的输出必须在一定的误差容忍度内（ `atol=5e-3, rtol=5e-3` ）与 `sparse_attn_ref` 的输出匹配。正确性不通过则为 0 分。
2. **性能**: 实现的**端到端运行时**（sparse_attn_triton）必须显著短于 `sparse_attn_ref` 的运行时。Triton 算子的性能会作为最终评奖的标准。
3. **高效利用硬件**:
    - 设计合理的 `BLOCK` 大小（如 `BLOCK_D` , `BLOCK_SEQ` , `BLOCK_Q_HEADS` , `BLOCK_PAGE_NUM` ），以匹配 NPU 的共享内存大小、寄存器数量和内存访问模式。
    - 可以使用 Triton 的 `autotune` 调优参数 autotune 参考链接
    - 你也可以使用**其他任何合理**的调优性能的方法，不做限制