NSA 稀疏注意力机制 by deepseek

1. NSA致力于实现硬件对齐的推理加速，通过特定的算法设计减少内存访问和硬件调度瓶颈， NSA 速度在 64k inference相较 Flash Attention 前向加速 9倍，反向加速 6倍
2. NSA的总体框架是通过更紧凑和信息密集的表示来替换原始的键值对
3. NSA有三种映射策略，分别是压缩（ cmp）、选择（ slc）和滑动窗口（ win ）。通过将不同策略得到的键值对进行组合

理解

1. 引入 动态 选择和压缩历史的 KV，减少计算量，符合实际的自然语言规律，但是
   1. 不一定完全匹配语言的表达逻辑
   2. 没有改变 transformer的固有问题，多层信息不共享等
2. 一定程度上等价于增加一层 attention，增加训练难度

原理

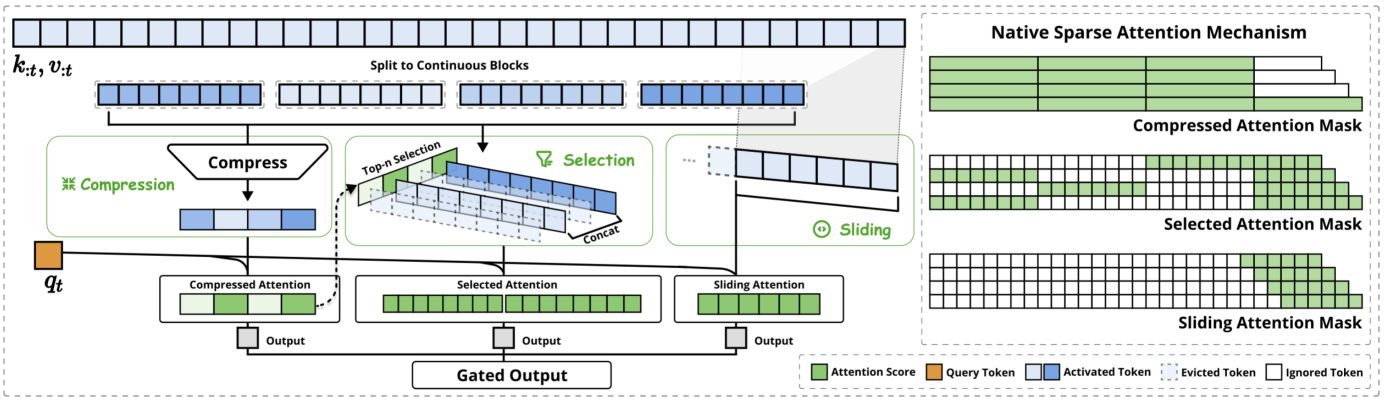

Figure 2 | Overview of NSA's architecture. Left: The framework processes input sequences through three parallel attention branches: For a given query, preceding keys and values are processed into compressed attention for coarse-grained patterns, selected attention for important token blocks, and sliding attention for local context. Right: Visualization of different attention patterns produced by each branch. Green areas indicate regions where attention scores need to be computed, while white areas represent regions that can be skipped.

假设上下文为 64k时 ，如果我们取 128个全局压缩 KV， 8个 512选择块 KV和就近窗口 4096个 KV，那么我们得到了压缩倍数 7.88

$$65536/(128 + 8 \times 512 + 4096) = 65536/8320 \approx 7.88$$

1. **tokens[illegible]** [illegible]

   1.
   ```
   W_K_cmp = torch.randn(l, 1) #MLP: W2[1,4l]@(W1[4l, l]@X[l, d])
   W_V_cmp = torch.randn(l, 1)
   W_pe = torch.randn(l, dim)

   K_cmp = []
   V_cmp = []
   for i in range(max_idx):
       cur_K = K[:, i * d + 0: i * d + l , :] + W_pe.unsqueeze(0)
       cur_V = V[:, i * d + 0: i * d + l , :] + W_pe.unsqueeze(0)
       cur_K = cur_K.transpose(1, 2) @ W_K_cmp
       cur_V = cur_V.transpose(1, 2) @ W_V_cmp
       K_cmp.append(cur_K)
       V_cmp.append(cur_V)

   K_cmp = torch.cat(K_cmp, dim = 2).transpose(1,2)
   V_cmp = torch.cat(V_cmp, dim = 2).transpose(1,2)
   print(K_cmp.shape) # torch.Size([1, 4, 16]) # [illegible] 32->4
   print(V_cmp.shape) # torch.Size([1, 4, 16]) # [illegible] 32->4
   ```

2. **tokens[illegible]** [illegible]

   1.
   ```
   idx_slc_start = idx * d
   idx_slc_end = idx * d + l
   K_slc = torch.randn(batch_size, t, d * select_top_k, dim)
   V_slc = torch.randn(batch_size, t, d * select_top_k, dim)
   for i in range(batch_size):
       for j in range(t):
           for k in range(select_top_k):
               K_slc[i, j, k * d : k * d + l, :] = K[i, idx_slc_start[i, j, k ] :  idx_slc_end[i, j, k ] , :]
               V_slc[i, j, k * d : k * d + l, :] = V[i, idx_slc_start[i, j, k ] :  idx_slc_end[i, j, k ] , :]
   print(K_slc.shape) # bs, seq_len, select_kv, dim, 1,32,16,16, [illegible] t[illegible] select_kv
   print(V_slc.shape) # bs, seq_len, select_kv, dim  1,32,16,16, [illegible] t[illegible] select_kv
   ```

3. [illegible] [illegible] tokens[illegible] NSA [illegible] context[illegible] q[illegible] kv [illegible] KV[illegible]

```
# built sliding window attention
def get_window_mask(seq_len, window):
    mask = torch.ones(seq_len, seq_len)
    mask = torch.tril(mask)
    win_mask = torch.ones(seq_len - window, seq_len - window)
    win_mask = 1.0 - torch.tril(win_mask)
    mask[window:, :seq_len - window] = win_mask
    return mask
print(get_window_mask(7, 3)) # test
window_mask = get_window_mask(t, 8)
```

4. 注意力聚合 ：在上述三个注意力计算中，我们都得到了同样维度 `[1, 32, 16]` 的注意力输出

```
o_list = [o_cmp, o_slc, o_win]
o_star = torch.zeros(batch_size, t, dim)
for i in range(3):
    o_star += gate[:, :, i].unsqueeze(2) * o_list[i]
print(o_star.shape)
```

计算加速

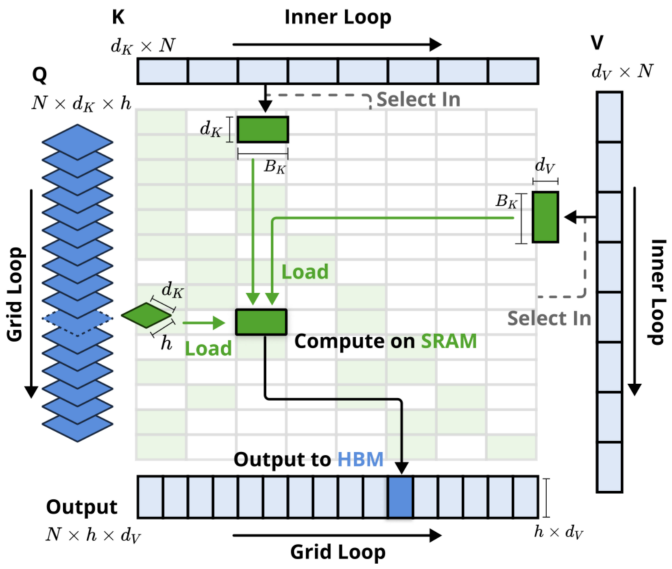

Figure 3 | Kernel design for NSA. The kernel loads queries by GQA groups (Grid Loop), fetches corresponding sparse KV blocks (Inner Loop), and performs attention computation on SRAM. Green blocks indicate data on SRAM, while blue indicates data on HBM.

Revision #5
Created 8 March 2025 08:42:58 by Colin
Updated 8 March 2025 09:44:34 by Colin