MSRA SH Triton Study Group
7. Triton Fused Kernels

2025/09/12

Problem #1: Fused SiLU

- Definition:

$$\mathrm{silu}(x, y) = \frac{xy}{1 + \exp(x)}$$

- Full element-wise function

```
Problem size: (4096, 4096)
         naive_silu  fused_silu  compiled_silu
latency    0.701396    0.129117       0.129992
```

```python
@triton.jit
def _fused_silu_kernel(
    x_ptr, y_ptr, M, N,
    stride_xm, stride_xn,
    stride_ym, stride_yn,
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr,
):
    pid_m = tl.program_id(axis=1)
    pid_n = tl.program_id(axis=0)
    offs_m = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_n = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    mask = offs_m < M
    offs_m %= M
    x1_ptrs = x_ptr + offs_m[:, None] * stride_xm + offs_n[None, :] * stride_xn
    x3_ptrs = x1_ptrs + N * stride_xn
    y_ptrs = y_ptr + offs_m[:, None] * stride_ym + offs_n[None, :] * stride_yn
    x1 = tl.load(x1_ptrs).to(tl.float32)
    x3 = tl.load(x3_ptrs).to(tl.float32)
    y = x1 * (1 / (1 + tl.exp(-x1))) * x3
    tl.store(y_ptrs, y.to(y_ptr.type.element_ty), mask=mask[:, None])
```

Problem #2: Fused MoE Routing

- Definition

```python
def moe_routing(
    hidden_states: torch.Tensor,
    gating_weight: torch.Tensor,
    num_experts: int,
    top_k: int,
):
    num_tokens, hidden_dim = hidden_states.shape
    router_logits = torch.nn.functional.linear(hidden_states, gating_weight, bias=None)
    router_logits = torch.nn.functional.softmax(router_logits, dim=1, dtype=torch.float32)
    routing_weights, selected_experts = torch.topk(router_logits, top_k, dim=-1)
    routing_weights = routing_weights.to(hidden_states.dtype)
    expert_mask = torch.nn.functional.one_hot(selected_experts.flatten(), num_classes=num_experts)
    expert_cnt = expert_mask.sum(dim=0)
    expert_idx = expert_mask.argsort(dim=0, descending=True)[:num_tokens].T.contiguous()
    return router_logits, routing_weights, expert_cnt, expert_idx, selected_experts
```

```python
@triton.jit
def _moe_routing_kernel(
    a_ptr,  # [M, K]
    b_ptr,  # [E, K]
    c_ptr,  # [M, E]
    w_ptr,  # [M, T]
    t_ptr,  # [M, T]
    cnt_ptr,  # [E] in [0, M)
    idx_ptr,  # [E, M] in [0, T * M)
    M, E, K, T,
    stride_am, stride_ak,
    stride_be, stride_bk,
    stride_cm, stride_ce,
    stride_wm, stride_we,
    stride_tm, stride_te,
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
    BLOCK_SIZE_E: tl.constexpr, BLOCK_SIZE_T: tl.constexpr,
):
    pid_m = tl.program_id(axis=0)  # pid_e = 0

    if pid_m * BLOCK_SIZE_M >= M:
        return

    offs_m = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_e = tl.arange(0, BLOCK_SIZE_E)
    offs_k = tl.arange(0, BLOCK_SIZE_K)

    a_ptrs = a_ptr + offs_m[:, None] * stride_am + offs_k[None, :] * stride_ak
    b_ptrs = b_ptr + offs_e[None, :] * stride_be + offs_k[:, None] * stride_bk

    # GeMM
    accumulator = tl.zeros([BLOCK_SIZE_M, BLOCK_SIZE_E], dtype=tl.float32)
    for k in range(0, K, BLOCK_SIZE_K):
        a = tl.load(a_ptrs)
        b = tl.load(b_ptrs)
        accumulator += tl.dot(a, b)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk

    # Softmax
    c_max = tl.max(accumulator, axis=1)
    c_exp = tl.math.exp(accumulator - c_max[:, None])
    c_sum = tl.sum(c_exp, axis=1)
    c = c_exp / c_sum[:, None]

    # Save Routing Weights
    offs_m = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    c_ptrs = c_ptr + offs_m[:, None] * stride_cm + offs_e[None, :] * stride_ce
    tl.store(c_ptrs, c.to(c_ptr.type.element_ty), mask=mask_m[:, None])

    # Top-K
    w_ptrs = w_ptr + offs_m * stride_wm
    t_ptrs = t_ptr + offs_m * stride_tm
    offs_t = tl.arange(0, BLOCK_SIZE_T)
    e_idx = tl.zeros([BLOCK_SIZE_M, BLOCK_SIZE_T], dtype=tl.int32)
    for k in tl.static_range(BLOCK_SIZE_T):
        max_c, max_idx = tl.max(c, axis=1, return_indices=True)
        c = tl.where(offs_e[None, :] == max_idx[:, None], 0.0, c)
        e_idx = tl.where(offs_t[None, :] == k, max_idx[:, None], e_idx)
        tl.store(w_ptrs, max_c.to(w_ptr.type.element_ty), mask=mask_m)
        tl.store(t_ptrs, max_idx.to(t_ptr.type.element_ty), mask=mask_m)
        w_ptrs += stride_we
        t_ptrs += stride_te

    # Histogram
    e_idx = tl.reshape(e_idx, [BLOCK_SIZE_M * BLOCK_SIZE_T, ])
    mask = tl.reshape(
        tl.broadcast_to(offs_m[:, None] < M, [BLOCK_SIZE_M, BLOCK_SIZE_T]),
        [BLOCK_SIZE_M * BLOCK_SIZE_T, ]
    )
    m_idx = tl.atomic_add(
        cnt_ptr + e_idx, tl.zeros_like(e_idx) + 1,
        mask=mask, sem='relaxed',
    )
    token_idx = tl.reshape(
        offs_m[:, None] * T + offs_t[None, :],
        [BLOCK_SIZE_M * BLOCK_SIZE_T, ],
    )
    tl.store(idx_ptr + e_idx * M + m_idx, token_idx, mask=mask)
```

Problem #2: Fused MoE Routing

- Complex workload
  - Matmul
  - Row reduce (softmax, top-K)
  - Custom reduce (atomic add)

- Latency

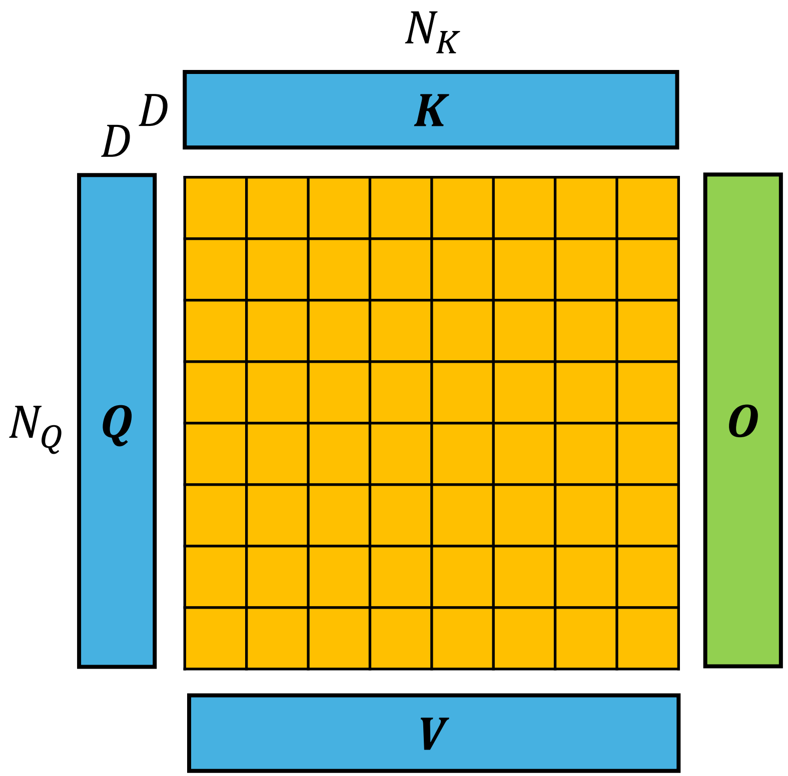
```
Problem size: (4096, 4096)
         naive_routing  fused_routing  compiled_routing
latency       2.095337       0.097784          2.028817
```

Problem #3: Sparse Attention

- Definition: add custom mask to attention kernels

Flash Attention

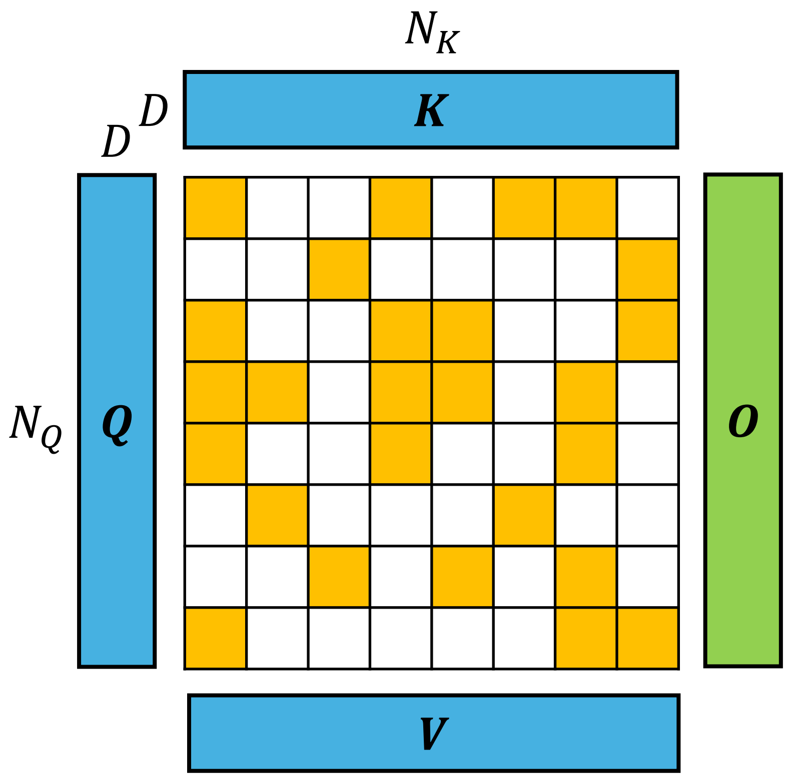
- Parallel for $1 \leq i \leq \frac{N_Q}{T_Q}$
  - Initialize $\boldsymbol{O}_i, M_i, L_i$
  - Load $\boldsymbol{Q}_i$ from HBM
  - For $1 \leq j \leq \frac{N_K}{T_K}$
    - Load $\boldsymbol{K}_j$ and $\boldsymbol{V}_j$ from HBM
    - $\boldsymbol{S}_{ij} \leftarrow \frac{\boldsymbol{Q}_i \boldsymbol{K}_j^{\mathrm{T}}}{\sqrt{D}}, M_i^{\mathrm{local}} \leftarrow \max_{-1}(\boldsymbol{S}_{ij})$
    - $M_i^{\mathrm{new}} \leftarrow \max(M_i, M_i^{\mathrm{local}}), \boldsymbol{P}_{ij} \leftarrow \frac{\exp(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}})}{L_i}, L_i^{\mathrm{local}} \leftarrow \mathrm{sum}_{-1}\left(\exp(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}})\right)$
    - $\alpha \leftarrow \exp(M_i - M_i^{\mathrm{new}}), L_i \leftarrow \alpha L_i + L_i^{\mathrm{local}}, M_i \leftarrow M_i^{\mathrm{new}}$
    - $\boldsymbol{O}_i \leftarrow \alpha \boldsymbol{O}_i + \boldsymbol{P}_{ij} \boldsymbol{V}_j$
  - Save $\boldsymbol{O}_i \leftarrow \frac{\boldsymbol{O}_i}{L_i}$ and $LSE_i \leftarrow M_i + \log(L_i)$ to HBM



```python
@triton.jit
def _attn_fwd_loop(
    q, k_ptrs, v_ptrs, o, m, l,
    offs_m, offs_n, stride_kn, stride_vn,
    start, end, BLOCK_SIZE_N: tl.constexpr,
):
    for start_n in range(start, end, BLOCK_SIZE_N):

        # Load K, V
        k = tl.load(k_ptrs + start_n * stride_kn)
        v = tl.load(v_ptrs + start_n * stride_vn)

        # Calc S <- Q @ K^T
        ...

        # Calc new row max M' <- max(M, max(s)), P <- exp(S - M'), local sum exp L1 <- sum(P)
        ...

        # Update L <- L * exp(M - M') + L1, M <- M'
        ...

        # Update O <- O * exp(M - M') + P @ V
        ...

    return o, m, l
```

Block-Sparse Attention

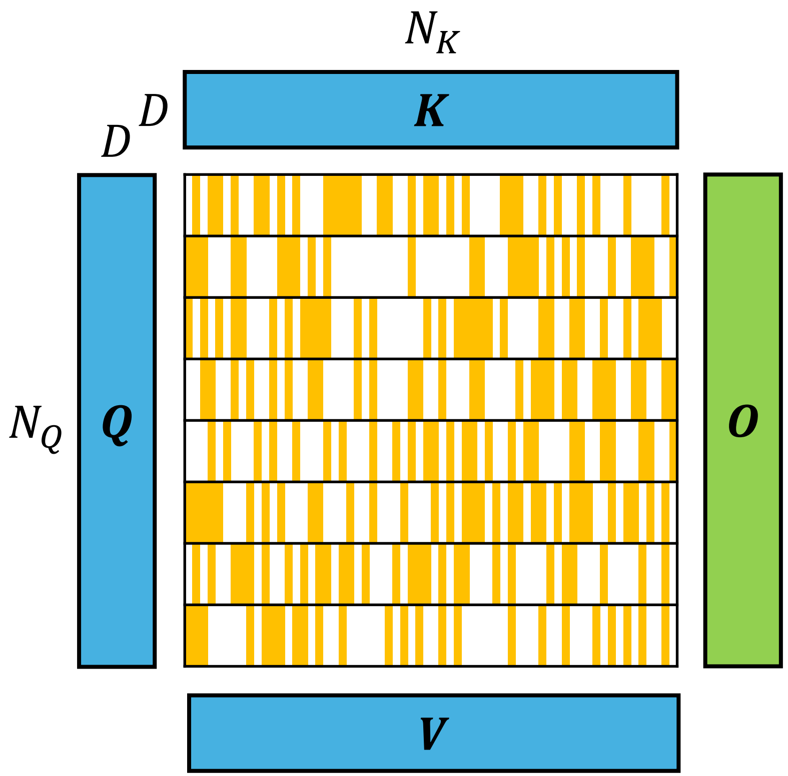
- Parallel for $1 \le i \le \frac{N_Q}{T_Q}$
  - Initialize $\boldsymbol{O}_i, M_i, L_i$
  - Load $\boldsymbol{Q}_i$ from HBM
  - For $1 \le j \le CNT_i$
    - Load $\boldsymbol{K}_{IDX_{ij}}$ and $\boldsymbol{V}_{IDX_{ij}}$ from HBM
    - $\boldsymbol{S}_{ij} \leftarrow \frac{\boldsymbol{Q}_i \boldsymbol{K}_j^{\mathrm{T}}}{\sqrt{D}}, M_i^{\mathrm{local}} \leftarrow \max_{-1}(\boldsymbol{S}_{ij})$
    - $M_i^{\mathrm{new}} \leftarrow \max\left(M_i, M_i^{\mathrm{local}}\right), \boldsymbol{P}_{ij} \leftarrow \frac{\exp\left(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}}\right)}{L_i}, L_i^{\mathrm{local}} \leftarrow \mathrm{sum}_{-1}\left(\exp\left(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}}\right)\right)$
    - $\alpha \leftarrow \exp(M_i - M_i^{\mathrm{new}}), L_i \leftarrow \alpha L_i + L_i^{\mathrm{local}}, M_i \leftarrow M_i^{\mathrm{new}}$
    - $\boldsymbol{O}_i \leftarrow \alpha \boldsymbol{O}_i + \boldsymbol{P}_{ij} \boldsymbol{V}_j$
  - Save $\boldsymbol{O}_i \leftarrow \frac{\boldsymbol{O}_i}{L_i}$ and $LSE_i \leftarrow M_i + \log(L_i)$ to HBM



```python
def _attn_fwd_loop(
    q, k_ptrs, v_ptrs, o, m, l,
    offs_m, offs_n, stride_kn, stride_vn,
    block_cnt, block_idx, BLOCK_SIZE_N: tl.constexpr,
):
    for j in range(0, block_cnt):

        # Load K, V
        start_n = tl.load(block_idx + j) * BLOCK_SIZE_N
        k = tl.load(k_ptrs + start_n * stride_kn)
        v = tl.load(v_ptrs + start_n * stride_vn)

        # Calc S <- Q @ K^T
        ...

        # Calc new row max M' <- max(M, max(s)), P <- exp(S - M'), local sum exp L1 <- sum(P)
        ...

        # Update L <- L * exp(M - M') + L1, M <- M'
        ...

        # Update O <- O * exp(M - M') + P @ V
        ...

    return o, m, l
```

PIT Attention

- Parallel for $1 \le i \le \frac{N_Q}{T_Q}$
  - Initialize $\boldsymbol{O}_i, M_i, L_i$
  - Load $\boldsymbol{Q}_i$ from HBM
  - For $1 \le j \le \frac{CNT_i}{T_K}$
    - Load $\boldsymbol{K}_{IDX_{i,jT_K:(j+1)T_K}}$ and $\boldsymbol{V}_{IDX_{i,jT_K:(j+1)T_K}}$ from HBM
    - $\boldsymbol{S}_{ij} \leftarrow \frac{\boldsymbol{Q}_i \boldsymbol{K}_j^{\mathrm{T}}}{\sqrt{D}}, M_i^{\mathrm{local}} \leftarrow \max_{-1}(\boldsymbol{S}_{ij})$
    - $M_i^{\mathrm{new}} \leftarrow \max(M_i, M_i^{\mathrm{local}}), \boldsymbol{P}_{ij} \leftarrow \frac{\exp(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}})}{L_i}, L_i^{\mathrm{local}} \leftarrow \mathrm{sum}_{-1}\left(\exp(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}})\right)$
    - $\alpha \leftarrow \exp(M_i - M_i^{\mathrm{new}}), L_i \leftarrow \alpha L_i + L_i^{\mathrm{local}}, M_i \leftarrow M_i^{\mathrm{new}}$
    - $\boldsymbol{O}_i \leftarrow \alpha \boldsymbol{O}_i + \boldsymbol{P}_{ij} \boldsymbol{V}_j$
  - Save $\boldsymbol{O}_i \leftarrow \frac{\boldsymbol{O}_i}{L_i}$ and $LSE_i \leftarrow M_i + \log(L_i)$ to HBM



```python
def _attn_fwd_loop(
    q, k_ptrs, v_ptrs, o, m, l,
    offs_m, offs_n, stride_kn, stride_vn,
    col_cnt, col_idx, BLOCK_SIZE_N: tl.constexpr,
):
    for start_n in range(0, col_cnt, BLOCK_SIZE_N):

        # Load K, V
        idx = tl.load(col_idx + start_n + offs_n)
        k = tl.load(k_ptrs + idx * stride_kn)
        v = tl.load(v_ptrs + idx * stride_vn)

        # Calc S <- Q @ K^T
        ...

        # Calc new row max M' <- max(M, max(s)), P <- exp(S - M'), local sum exp L1 <- sum(P)
        ...

        # Update L <- L * exp(M - M') + L1, M <- M'
        ...

        # Update O <- O * exp(M - M') + P @ V
        ...

    return o, m, l
```

Reading Materials

- Introduction to torch.compile — PyTorch Tutorials 2.8.0+cu128 documentation
- Layer Normalization — Triton documentation
- [2407.02490] MInference 1.0: Accelerating Pre-filling for Long-Context LLMs via Dynamic Sparse Attention