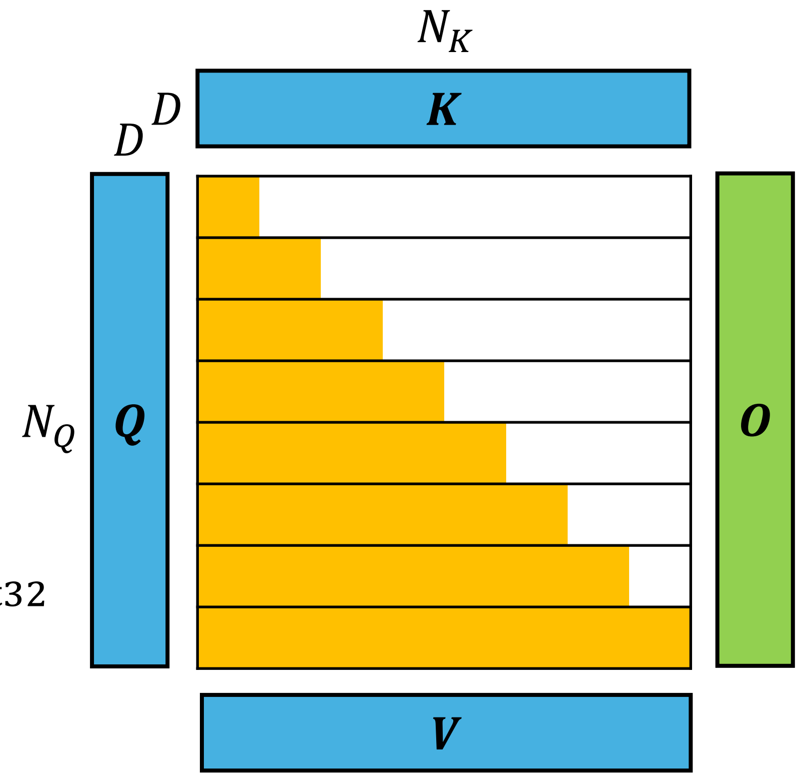
MSRA SH Triton Study Group
6. Triton Flash-Attention

2025/09/05

Flash-Attention 2 Forward

- Tile $\boldsymbol{Q}, \boldsymbol{O}$ into $\frac{N_Q}{T_Q}$ blocks $\boldsymbol{Q}_i, \boldsymbol{O}_i$
- Tile $\boldsymbol{K}, \boldsymbol{V}$ into $\frac{N_K}{T_K}$ blocks $\boldsymbol{K}_j, \boldsymbol{V}_j$
- Parallel for $1 \le i \le \frac{N_Q}{T_Q}$
  - in Registers: $\boldsymbol{O}_i \leftarrow (0)^{T_Q \times D}_{\mathrm{float32}}, M_i \leftarrow (-\infty)^{T_Q}_{\mathrm{float32}}, L_i \leftarrow (0)^{T_Q}_{\mathrm{float32}}$
  - For $1 \le j \le \frac{N_K}{T_K}$
    - $\boldsymbol{S}_{ij} \leftarrow \frac{\boldsymbol{Q}_i \boldsymbol{K}_j^{\mathrm{T}}}{\sqrt{D}}, M_i^{\mathrm{local}} \leftarrow \max_{-1}(\boldsymbol{S}_{ij})$
    - $M_i^{\mathrm{new}} \leftarrow \max(M_i, M_i^{\mathrm{local}}), \boldsymbol{P}_{ij} \leftarrow \frac{\exp(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}})}{L_i}, L_i^{\mathrm{local}} \leftarrow \mathrm{sum}_{-1}\left(\exp(\boldsymbol{S}_{ij} - M_i^{\mathrm{new}})\right)$
    - $\alpha \leftarrow \exp(M_i - M_i^{\mathrm{new}}), L_i \leftarrow \alpha L_i + L_i^{\mathrm{local}}, M_i \leftarrow M_i^{\mathrm{new}}$
    - $\boldsymbol{O}_i \leftarrow \alpha \boldsymbol{O}_i + \boldsymbol{P}_{ij} \boldsymbol{V}_j$
  - Save $\boldsymbol{O}_i \leftarrow \frac{\boldsymbol{O}_i}{L_i}$ (to float16) and $LSE_i \leftarrow M_i + \log(L_i)$ to GPU HBM



```python
batch_size, num_tokens, num_heads, head_dim = q.shape
batch_size_k, num_k_tokens, num_k_heads, head_dim_k = k.shape
batch_size_v, num_v_tokens, num_v_heads, head_dim_v = v.shape
assert batch_size == batch_size_k and batch_size_k == batch_size_v
assert num_tokens == num_k_tokens and num_k_tokens == num_v_tokens
assert num_heads == num_k_heads and num_k_heads == num_v_heads
assert head_dim == head_dim_k and head_dim_k == head_dim_v
assert head_dim in {16, 32, 64, 128}

sm_scale = sm_scale or head_dim ** (-0.5)
o = torch.empty_like(q)
lse = torch.empty((batch_size, num_heads, num_tokens), dtype=torch.float32, device=q.device)

grid = lambda META: (triton.cdiv(num_tokens, META['BLOCK_SIZE_M']), num_heads * batch_size, 1)
flash_attn_fwd_kernel[grid](
    q, k, v, o, lse,
    q.stride(0), q.stride(2), q.stride(1), q.stride(3),
    k.stride(0), k.stride(2), k.stride(1), k.stride(3),
    v.stride(0), v.stride(2), v.stride(1), v.stride(3),
    o.stride(0), o.stride(2), o.stride(1), o.stride(3),
    lse.stride(0), lse.stride(1), lse.stride(2),
    sm_scale, num_tokens, num_heads,
    CAUSAL=causal, BLOCK_SIZE_D=head_dim,
)
```

```python
@triton.jit
def flash_attn_fwd_kernel(
    q_ptr, k_ptr, v_ptr, o_ptr, lse_ptr,
    stride_qb, stride_qh, stride_qm, stride_qd,
    stride_kb, stride_kh, stride_kn, stride_kd,
    stride_vb, stride_vh, stride_vn, stride_vd,
    stride_ob, stride_oh, stride_om, stride_od,
    stride_lb, stride_lh, stride_lm,
    sm_scale, num_tokens, num_heads, CAUSAL: tl.constexpr,
    BLOCK_SIZE_M: tl.constexpr,
    BLOCK_SIZE_N: tl.constexpr,
    BLOCK_SIZE_D: tl.constexpr,
):
    start_m = tl.program_id(0) * BLOCK_SIZE_M
    off_h = tl.program_id(1) % num_heads
    off_b = tl.program_id(1) // num_heads

    # Initialize offsets
    offs_m = start_m + tl.arange(0, BLOCK_SIZE_M)
    offs_n = tl.arange(0, BLOCK_SIZE_N)
    offs_d = tl.arange(0, BLOCK_SIZE_D)
    mask_m = offs_m < num_tokens
    q_ptrs = q_ptr + off_b * stride_qb + off_h * stride_qh + offs_m[:, None] * stride_qm + offs_d[None, :] * stride_qd
    k_ptrs = k_ptr + off_b * stride_kb + off_h * stride_kh + offs_n[:, None] * stride_kn + offs_d[None, :] * stride_kd
    v_ptrs = v_ptr + off_b * stride_vb + off_h * stride_vh + offs_n[:, None] * stride_vn + offs_d[None, :] * stride_vd
    o_ptrs = o_ptr + off_b * stride_ob + off_h * stride_oh + offs_m[:, None] * stride_om + offs_d[None, :] * stride_od
    lse_ptrs = lse_ptr + off_b * stride_lb + off_h * stride_lh + offs_m * stride_lm
```

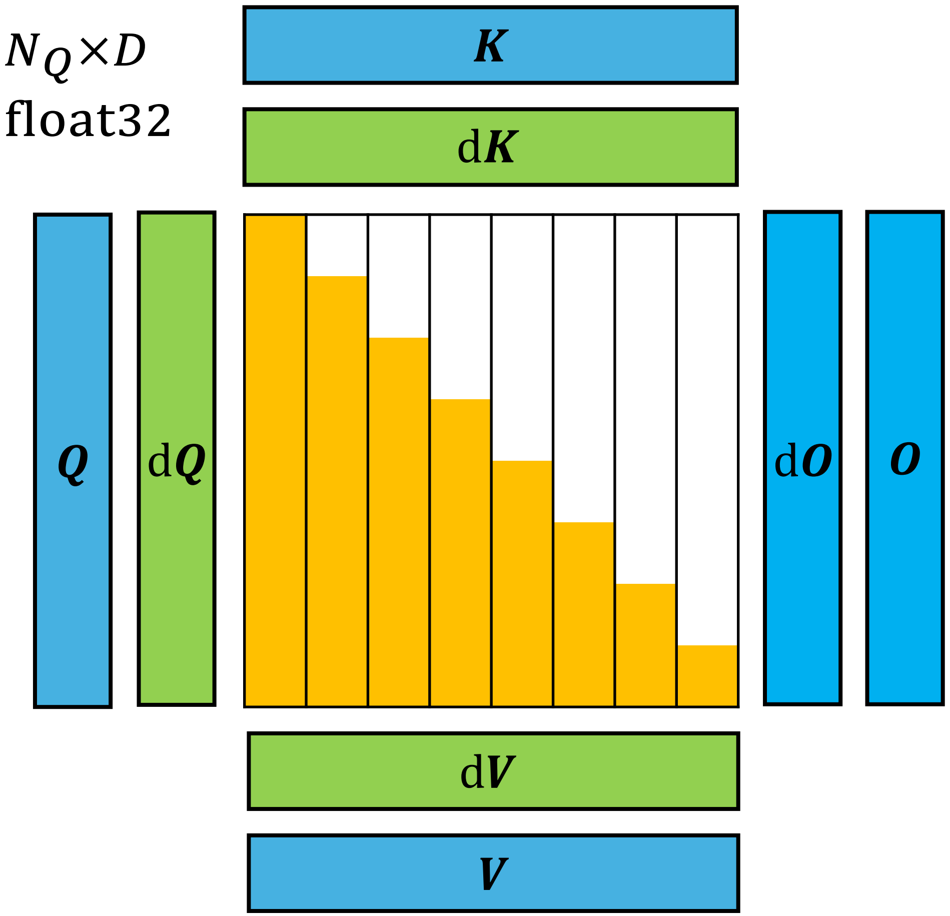
```python
m = tl.zeros([BLOCK_SIZE_M], dtype=tl.float32) - float("inf")
l = tl.zeros([BLOCK_SIZE_M], dtype=tl.float32)
o = tl.zeros([BLOCK_SIZE_M, BLOCK_SIZE_D], dtype=tl.float32)
q = tl.load(q_ptrs, mask=mask_m[:, None], other=0.0)

# Scale sm_scale by log_2(e) and use 2^x instead of exp
q = (q * sm_scale * 1.4426950408889634).to(q_ptr.type.element_ty)

# Split the main loop into 2 parts
if CAUSAL:
    start, mid, end = 0, start_m // BLOCK_SIZE_N * BLOCK_SIZE_N, tl.minimum(start_m + BLOCK_SIZE_M, num_tokens)
else:
    start, mid, end = 0, num_tokens // BLOCK_SIZE_N * BLOCK_SIZE_N, num_tokens

# Main loop part 1: no mask at first
o, m, l = _attn_fwd_loop(
    q, k_ptrs, v_ptrs, o, m, l,
    offs_m, offs_n, stride_kn, stride_vn,
    start, mid, BLOCK_SIZE_N=BLOCK_SIZE_N,
    CAUSAL=False, MASK_N=False,
)
# Main loop part 2: causal mask and kv data mask for last blocks
o, m, l = _attn_fwd_loop(
    q, k_ptrs, v_ptrs, o, m, l,
    offs_m, offs_n, stride_kn, stride_vn,
    mid, end, BLOCK_SIZE_N=BLOCK_SIZE_N,
    CAUSAL=CAUSAL, MASK_N=True,
)
```

```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,
    CAUSAL: tl.constexpr, MASK_N: tl.constexpr,
):
    for start_n in range(start, end, BLOCK_SIZE_N):

        # Load K, V
        ...

        # Calc S <- Q @ K^T, mask S if causal
        ...

        # 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
```

```python
    # Load K, V
    if MASK_N:
        mask_n = start_n + offs_n < end
        k = tl.load(k_ptrs + start_n * stride_kn, mask=mask_n[:, None], other=0.0)
        v = tl.load(v_ptrs + start_n * stride_vn, mask=mask_n[:, None], other=0.0)
    else:
        k = tl.load(k_ptrs + start_n * stride_kn)
        v = tl.load(v_ptrs + start_n * stride_vn)

    # Calc S <- Q @ K^T, mask S if causal
    s = tl.dot(q, k.T)
    if CAUSAL:
        s = tl.where(offs_m[:, None] >= start_n + offs_n[None, :], s, float("-inf"))
    elif MASK_N:
        s = tl.where(mask_n[None, :], s, float("-inf"))

    # Calc new row max M' <- max(M, max(s)), P <- exp(S - M'), local sum exp L1 <- sum(P)
    m_new = tl.maximum(m, tl.max(s, 1))
    p = tl.math.exp2(s - m_new[:, None])
    l_local = tl.sum(p, 1)

    # Update L <- L * exp(M - M') + L1, M <- M'
    alpha = tl.math.exp2(m - m_new)
    l = l * alpha + l_local
    m = m_new

    # Update O <- O * exp(M - M') + P @ V
    o = o * alpha[:, None] + tl.dot(p.to(v.type.element_ty), v)
```

Flash-Attention 2 Backward

- Preprocess: $\Delta_i = \sum_{k=0}^{D} \mathrm{d}\boldsymbol{O}_{ik} \odot \boldsymbol{O}_{ik}$, $\mathrm{d}\boldsymbol{Q} \leftarrow (0)_{\mathrm{float32}}^{N_Q \times D}$
- Parallel for $1 \leq j \leq \frac{N_K}{T_K}$
  - in Registers: $\mathrm{d}\boldsymbol{K}_j \leftarrow (0)_{\mathrm{float32}}^{T_K \times D}$, $\mathrm{d}\boldsymbol{V}_j \leftarrow (0)_{\mathrm{float32}}^{T_K \times D}$
  - For $1 \leq i \leq \frac{N_Q}{T_Q}$
    - $\boldsymbol{P}_{ij} \leftarrow \exp\left(\frac{\boldsymbol{Q}_i \boldsymbol{K}_j^{\mathrm{T}}}{\sqrt{D}} - LSE_i\right)$, $\mathrm{d}\boldsymbol{V}_j \leftarrow \mathrm{d}\boldsymbol{V}_j + \boldsymbol{P}_{ij}^{\mathrm{T}} \mathrm{d}\boldsymbol{O}_i$
    - $\mathrm{d}\boldsymbol{S}_{ij} \leftarrow \boldsymbol{P}_{ij} \odot \left(\mathrm{d}\boldsymbol{O}_i \boldsymbol{V}_j^{\mathrm{T}} - \Delta_i\right)$, $\mathrm{d}\boldsymbol{K}_j \leftarrow \mathrm{d}\boldsymbol{K}_j + \frac{\mathrm{d}\boldsymbol{S}_{ij}^{\mathrm{T}} \boldsymbol{Q}_i}{\sqrt{D}}$
    - $\mathrm{d}\boldsymbol{Q}_i \leftarrow \mathrm{d}\boldsymbol{Q}_i + \frac{\mathrm{d}\boldsymbol{S}_{ij} \boldsymbol{K}_j}{\sqrt{D}}$ by atomic_add()
  - Save $\mathrm{d}\boldsymbol{K}_j$ and $\mathrm{d}\boldsymbol{V}_j$ to GPU HBM (to float16)
- Convert $\mathrm{d}\boldsymbol{Q}$ to float16



Preprocess: $\Delta_i = \sum_{k=0}^{D} \mathrm{d}\boldsymbol{O}_{ik} \odot \boldsymbol{O}_{ik}$

```python
@torch.compile
def flash_attn_bwd_calc_delta(
    o: torch.Tensor,   # [batch_size, num_tokens, num_heads, head_dim]
    do: torch.Tensor,  # [batch_size, num_tokens, num_heads, head_dim]
):
    return torch.sum((o * do).swapaxes(1, 2), dim=-1, dtype=torch.float32)
```

```python
class TritonFlashAttention(torch.autograd.Function):

    @staticmethod
    def forward(
        ctx: torch.autograd.function.FunctionCtx,
        q: torch.Tensor,
        k: torch.Tensor,
        v: torch.Tensor,
        sm_scale: float,
        causal: bool,
    ):
        batch_size, num_tokens, num_heads, head_dim = q.shape

        # Check shapes
        ...

        # Launch forward kernel
        o, lse = ...

        ctx.batch_size = batch_size
        ctx.num_heads = num_heads
        ctx.num_tokens = num_tokens
        ctx.head_dim = head_dim
        ctx.sm_scale = sm_scale
        ctx.causal = causal
        ctx.save_for_backward(q, k, v, o, lse)

        return o
```

```python
    @staticmethod
    def backward(ctx, do: torch.Tensor):
        q, k, v, o, lse = ctx.saved_tensors

        delta = flash_attn_bwd_calc_delta(o, do)
        dq = torch.zeros_like(q, dtype=torch.float32)
        dk = torch.empty_like(k)
        dv = torch.empty_like(v)

        grid = lambda META: (
            triton.cdiv(ctx.num_tokens, META['BLOCK_SIZE_N']),
            ctx.num_heads * ctx.batch_size,
        )
        flash_attn_bwd_kernel[grid](
            q, k, v, lse, delta, dq, dk, dv, do,
            q.stride(0), q.stride(2), q.stride(1), q.stride(3),
            k.stride(0), k.stride(2), k.stride(1), k.stride(3),
            v.stride(0), v.stride(2), v.stride(1), v.stride(3),
            o.stride(0), o.stride(2), o.stride(1), o.stride(3),
            lse.stride(0), lse.stride(1), lse.stride(2),
            delta.stride(0), delta.stride(1), delta.stride(2),
            ctx.sm_scale, ctx.num_tokens, ctx.num_heads,
            CAUSAL=ctx.causal, BLOCK_SIZE_D=ctx.head_dim,
            SKIP_DQ=ctx.split_bwd,
        )

        return dq.to(q.dtype), dk, dv, None, None, None
```

```python
@triton.jit
def flash_attn_bwd_kernel(
    q_ptr, k_ptr, v_ptr, lse_ptr, delta_ptr,
    dq_ptr, dk_ptr, dv_ptr, do_ptr,
    stride_qb, stride_qh, stride_qm, stride_qd,
    stride_kb, stride_kh, stride_kn, stride_kd,
    stride_vb, stride_vh, stride_vn, stride_vd,
    stride_ob, stride_oh, stride_om, stride_od,
    stride_lb, stride_lh, stride_lm,
    stride_db, stride_dh, stride_dm,
    sm_scale, num_tokens, num_heads, CAUSAL: tl.constexpr,
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_D: tl.constexpr,
):
    start_n = tl.program_id(0) * BLOCK_SIZE_N
    off_h = tl.program_id(1) % num_heads
    off_b = tl.program_id(1) // num_heads

    # Initialize offsets
    offs_m = tl.arange(0, BLOCK_SIZE_M)
    offs_n = start_n + tl.arange(0, BLOCK_SIZE_N)
    offs_d = tl.arange(0, BLOCK_SIZE_D)
    mask_n = offs_n < num_tokens
    q_ptrs = q_ptr + off_b * stride_qb + off_h * stride_qh + offs_m[:, None] * stride_qm + offs_d[None, :] * stride_qd
    k_ptrs = k_ptr + off_b * stride_kb + off_h * stride_kh + offs_n[:, None] * stride_kn + offs_d[None, :] * stride_kd
    v_ptrs = v_ptr + off_b * stride_vb + off_h * stride_vh + offs_n[:, None] * stride_vn + offs_d[None, :] * stride_vd
    dq_ptrs = dq_ptr + off_b * stride_qb + off_h * stride_qh + offs_m[None, :] * stride_qm + offs_d[:, None] * stride_qd
    dk_ptrs = dk_ptr + off_b * stride_kb + off_h * stride_kh + offs_n[:, None] * stride_kn + offs_d[None, :] * stride_kd
    dv_ptrs = dv_ptr + off_b * stride_vb + off_h * stride_vh + offs_n[:, None] * stride_vn + offs_d[None, :] * stride_vd
    do_ptrs = do_ptr + off_b * stride_ob + off_h * stride_oh + offs_m[:, None] * stride_om + offs_d[None, :] * stride_od
    lse_ptrs = lse_ptr + off_b * stride_lb + off_h * stride_lh + offs_m * stride_lm
    delta_ptrs = delta_ptr + off_b * stride_db + off_h * stride_dh + offs_m * stride_dm
```

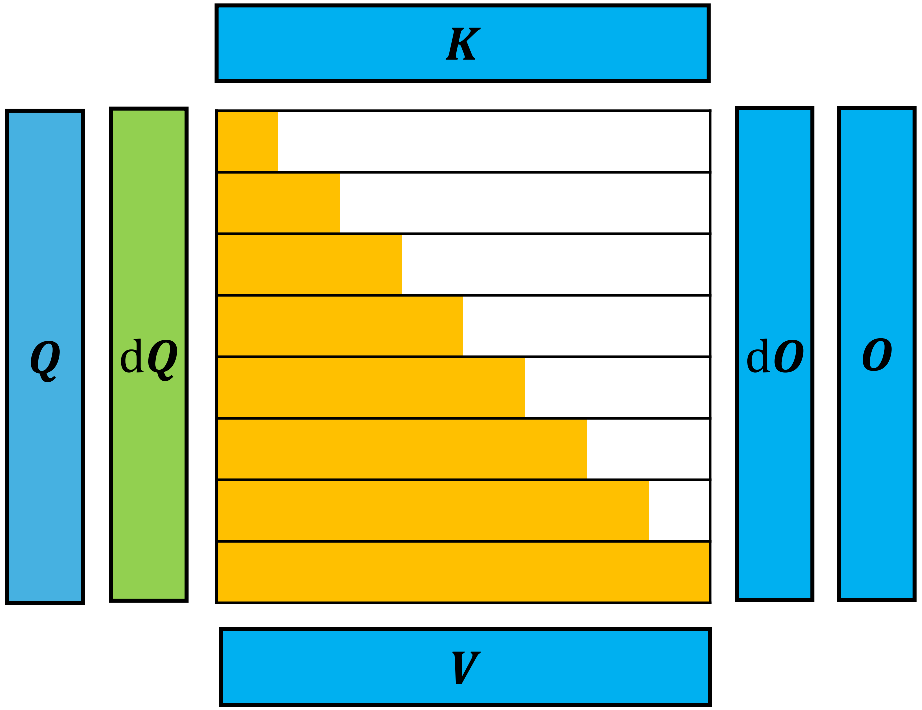
```python
dk = tl.zeros([BLOCK_SIZE_N, BLOCK_SIZE_D], dtype=tl.float32)
dv = tl.zeros([BLOCK_SIZE_N, BLOCK_SIZE_D], dtype=tl.float32)
k = (tl.load(k_ptrs, mask=mask_n[:, None], other=0.0) * (sm_scale * 1.4426950408889634)).to(k_ptr.type.element_ty)
v = tl.load(v_ptrs, mask=mask_n[:, None], other=0.0)

# Split the main loop into 3 parts
if CAUSAL:
    causal_start = start_n // BLOCK_SIZE_M * BLOCK_SIZE_M
    causal_end = tl.minimum(causal_start + tl.maximum(BLOCK_SIZE_M, BLOCK_SIZE_N), num_tokens)
    full_start = causal_end
    full_end = num_tokens // BLOCK_SIZE_M * BLOCK_SIZE_M
    mask_start = tl.maximum(full_end, causal_end)
    mask_end = num_tokens
else:
    causal_start, causal_end = 0, 0
    full_start, full_end = 0, num_tokens // BLOCK_SIZE_M * BLOCK_SIZE_M
    mask_start, mask_end = full_end, num_tokens

# Main loop part 3: mask last rows that exceed the sequence length
dk, dv = _attn_bwd_loop(mask_start, mask_end, CAUSAL=False, MASK_M=True)
# Main loop part 2: no mask
dk, dv = _attn_bwd_loop(full_start, full_end, CAUSAL=False, MASK_M=False)
# Main loop part 1: causal mask
dk, dv = _attn_bwd_loop(causal_start, causal_end, CAUSAL=True, MASK_M=True)

# Write back dK and dV
tl.store(dk_ptrs, (dk * sm_scale).to(dk_ptr.type.element_ty), mask=mask_n[:, None])
tl.store(dv_ptrs, dv.to(dv_ptr.type.element_ty), mask=mask_n[:, None])
```

```python
@triton.jit
def _attn_bwd_loop(
    k, v, dk, dv, dq_ptrs, q_ptrs, do_ptrs, lse_ptrs, delta_ptrs,
    offs_m, offs_n, stride_qm, stride_om, stride_lm, stride_dm,
    start, end, BLOCK_SIZE_M: tl.constexpr, CAUSAL: tl.constexpr, MASK_M: tl.constexpr,
):
    # Reverse loop to maximize L2 cache hit rate when causal
    for start_m in range(tl.cdiv(end, BLOCK_SIZE_M) * BLOCK_SIZE_M - BLOCK_SIZE_M, start - BLOCK_SIZE_M, -BLOCK_SIZE_M):

        # Load Q, dO, LSE (= M + log(L)), Δ (= sum(dO * O))
        ...

        # Calc P <- exp(Q @ K^T - M) / L
        ...

        # Update dV <- dV + P^T @ dO
        ...

        # Calc dS <- P * (dO @ V^T - Δ)
        ...

        # Update dQ <- dQ + dS @ K by atomic_add
        ...

        # Update dK <- dK + dS^T @ Q
        ...

    return dk, dv
```

```python
# Load Q, dO, LSE (= M + log(L)), Δ (= sum(dO * O))
q = tl.load(q_ptrs + start_m * stride_qm)
do = tl.load(do_ptrs + start_m * stride_om)
lse = tl.load(lse_ptrs + start_m * stride_lm)
delta = tl.load(delta_ptrs + start_m * stride_dm)

# Calc P <- exp(Q @ K^T - M) / L
pT = tl.math.exp2(tl.dot(k, q.T) - lse[None, :])
if CAUSAL:
    causal_mask = start_m + offs_m[None, :] >= offs_n[:, None]
    pT = tl.where(causal_mask, pT, 0.0)

# Update dV <- dV + P^T @ dO
dv += tl.dot(pT.to(do.type.element_ty), do)

# Calc dS <- P * (dO @ V^T - Δ)
dsT = (pT * (tl.dot(v, do.T) - delta[None, :])).to(q.type.element_ty)

# Update dQ <- dQ + dS @ K by atomic_add
dqT = tl.dot(k.T, dsT.to(k.type.element_ty)) * 0.6931471824645996
tl.atomic_add(dq_ptrs + start_m * stride_om, dqT, sem='relaxed')

# Update dK <- dK + dS^T @ Q
dk += tl.dot(dsT.to(q.type.element_ty), q)
```

Calculate $\mathbf{d}\boldsymbol{Q}$ separately using a new kernel

- Possible reasons:
  - Triton atomic_add() not good
  - Triton compile is not optimized
- Parallel for $1 \le i \le \frac{N_Q}{T_Q}$
  - in Registers: $\mathrm{d}\boldsymbol{Q}_i \leftarrow (0)^{T_Q \times D}_{\mathtt{float32}}$
  - For $1 \le j \le \frac{N_K}{T_K}$
    - $\boldsymbol{P}_{ij} \leftarrow \exp\left(\frac{\boldsymbol{Q}_i\boldsymbol{K}_j^{\mathrm{T}}}{\sqrt{D}} - LSE_i\right)$
    - $\mathrm{d}\boldsymbol{S}_{ij} \leftarrow \boldsymbol{P}_{ij} \odot \left(\mathrm{d}\boldsymbol{O}_i\boldsymbol{V}_j^{\mathrm{T}} - \Delta_i\right)$
    - $\mathrm{d}\boldsymbol{Q}_i \leftarrow \mathrm{d}\boldsymbol{Q}_i + \frac{\mathrm{d}\boldsymbol{S}_{ij}\boldsymbol{K}_j}{\sqrt{D}}$
  - Save $\mathrm{d}\boldsymbol{Q}_i$ to GPU HBM



Latency (Triton 3.2.0)

- Forward
  - Much faster than torch
  - 10%~15% slower than official
- Backward
  - atomic_add: ~80% slower
  - split_bwd: ~40% slower

```
=================================================================
Problem size: (1, 4096, 32, 128); Causal = False
                         fwd         bwd
triton_atomic_add    1.653951    7.951271
triton_split_bwd     1.661846    5.531596
flash_official       1.450124    4.171456
torch_naive         12.335835   19.094136
-----------------------------------------------------------------
Problem size: (1, 4096, 32, 128); Causal = True
                         fwd         bwd
triton_atomic_add    0.968440    4.335084
triton_split_bwd     0.968935    3.191221
flash_official       0.838765    2.319285
torch_naive         16.054613   24.706221
-----------------------------------------------------------------
Problem size: (1, 4321, 32, 128); Causal = False
                         fwd         bwd
triton_atomic_add    1.947054    8.969808
triton_split_bwd     1.945600    6.765008
flash_official       1.681893    4.926344
torch_naive         21.459967   31.271936
-----------------------------------------------------------------
Problem size: (1, 4321, 32, 128); Causal = True
                         fwd         bwd
triton_atomic_add    1.082380    4.805748
triton_split_bwd     1.092376    3.625090
flash_official       0.946758    2.642283
torch_naive         26.221909   37.962410
=================================================================
```

Homework

- Play with [TritonStudyGroup/6_Triton_Flash_Attn at main · Starmys/TritonStudyGroup]

- Support Grouped-Query Attention
- Implement Triton block-sparse Flash-Attention with block size of 128
- Output the attention scores **after pooling**

- Why do we split the bwd main loop into 3 parts (but not 2 parts)?
- Profile Triton flash-attention forward and explain why it is slow
- Calculate the FLOPs of `triton_split_bwd`

Reading Materials

- Triton flash attention example: Fused Attention — Triton documentation