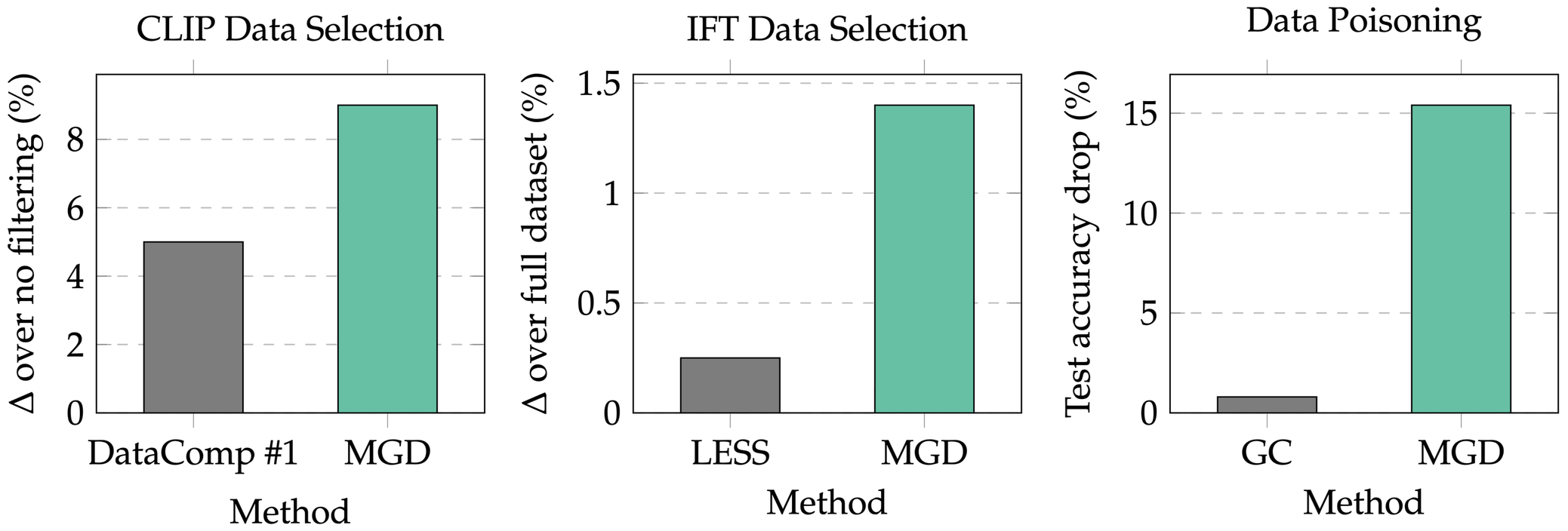
Flash-HOG

Higher-order gradients at lightspeed!

Marcel Roed, Neil Band, Herman Brunborg, Stephen Ge, Shiv Sundram

Why higher-order gradients?

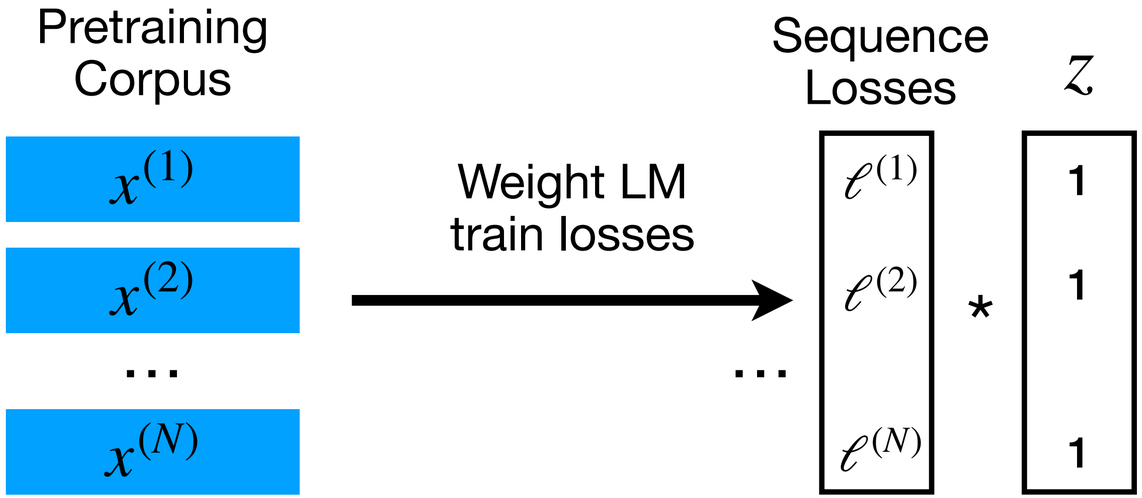
- State-of-the-art approach for pretraining/SFT data selection



Optimizing ML Training with Metagradient Descent. Engstrom et al., 2024.

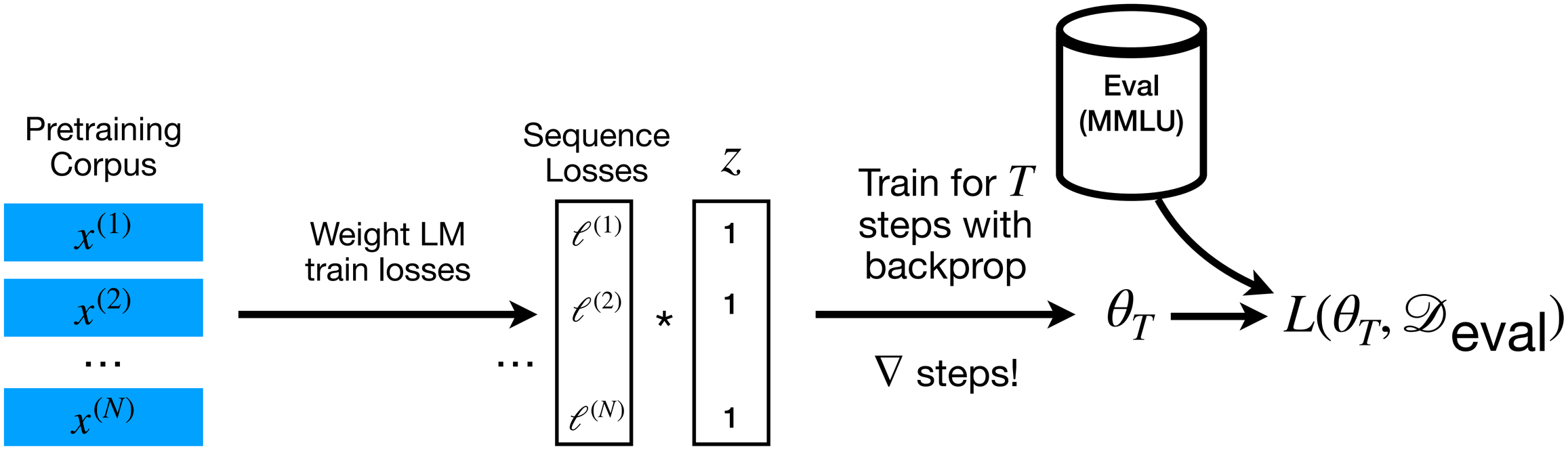
Why higher-order gradients: data selection

1. Weight per-sequence train losses with scalar *data weights* z



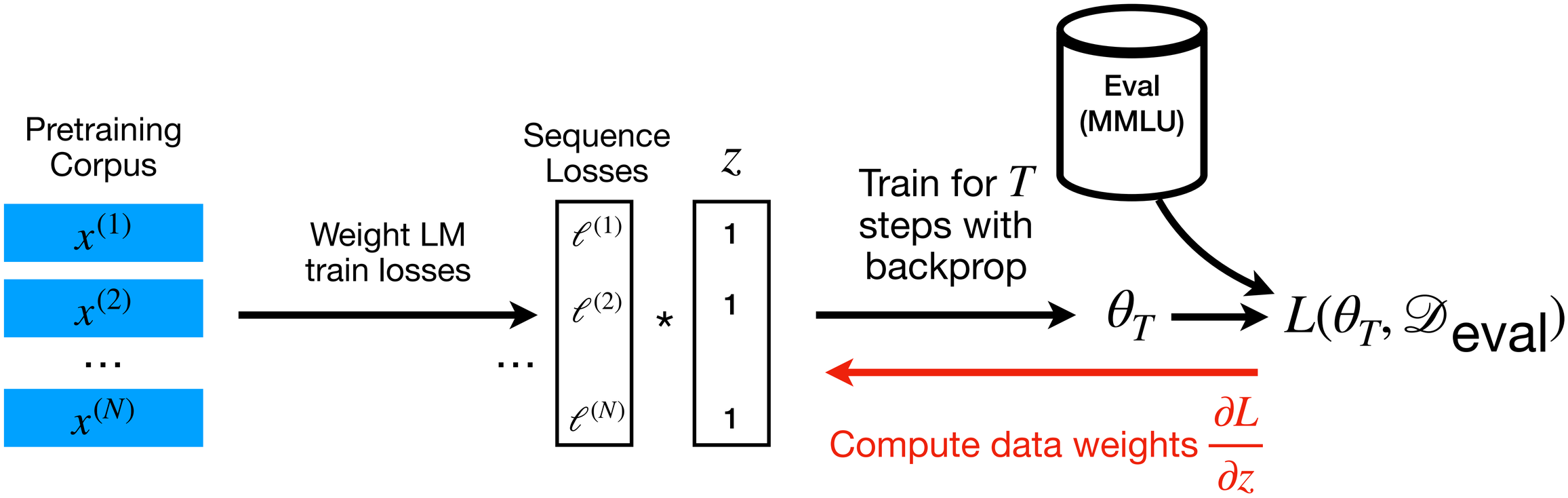
Why higher-order gradients: data selection

1. Weight per-sequence train losses with scalar *data weights* z



Why higher-order gradients: data selection

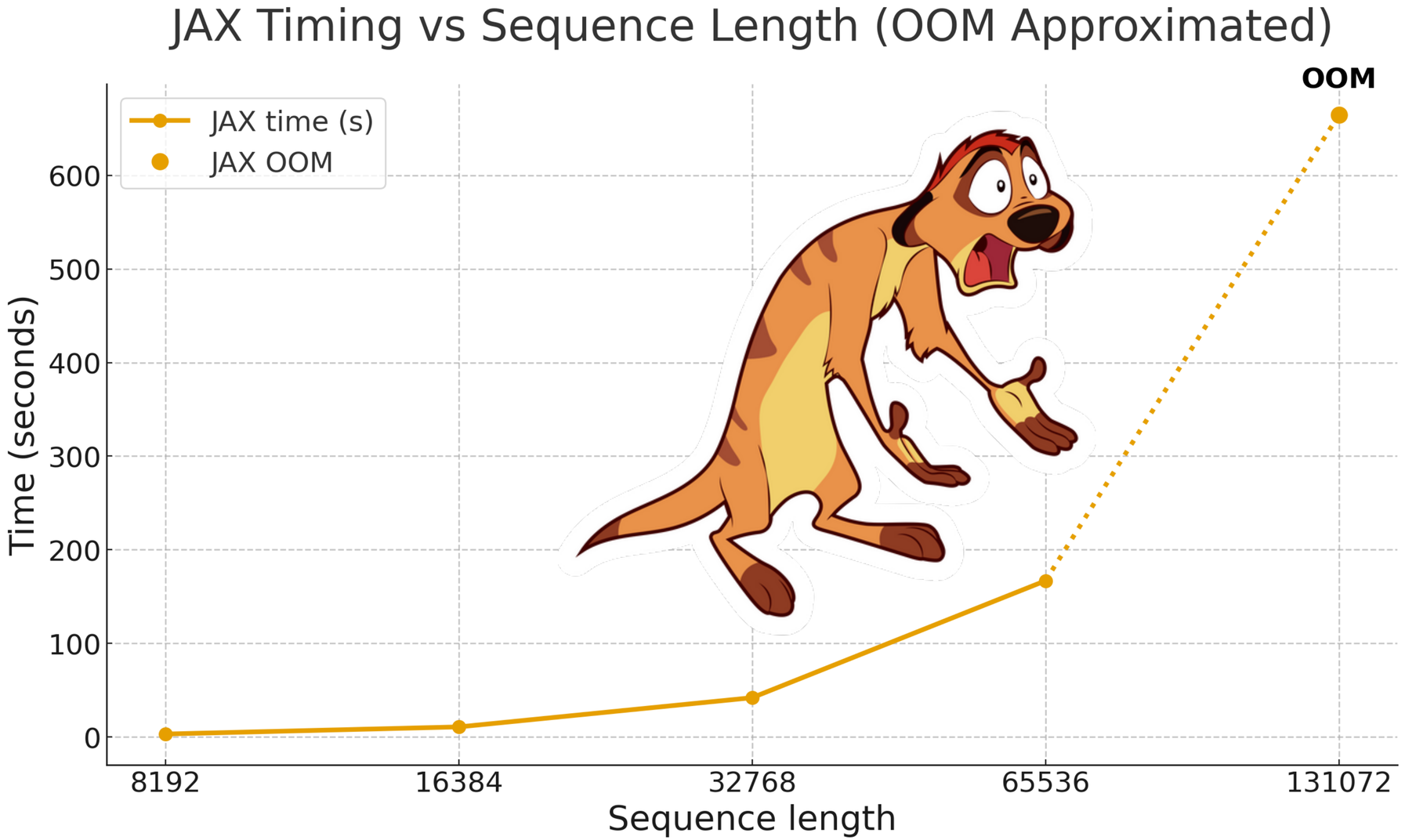
1. Weight per-sequence train losses with scalar *data weights* z
2. Differentiate a validation loss w.r.t. z to get "data importance weights"



Challenge:

Requires grad-on-grad,
which is naively super slow and memory-intensive!

Naive impl: OOM@131K seqlen (on B200)

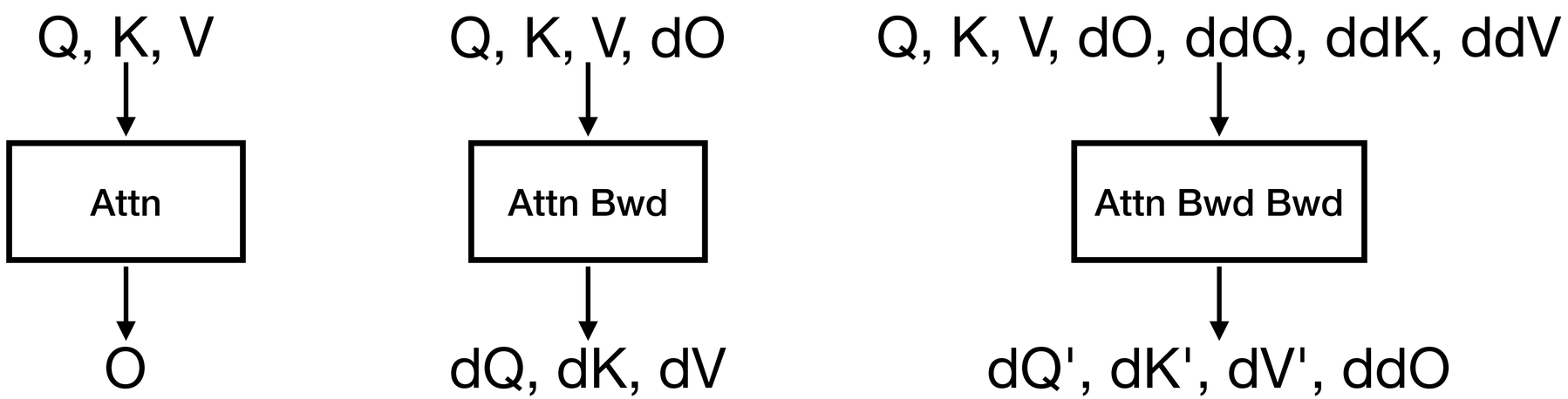


Naive impl: @1M seqlen

```
Allocator (GPU_0_bfc) ran out of memory trying to allocate 4.00TiB (rounded to 4398046511104)requested by op
```

4.00TiB !!!

Higher Order Backward



Attn

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top / \sqrt{d}$$

$$\mathbf{P}_{ij} = \mathrm{softmax}_j(\mathbf{S})_{ij}$$

$$\mathbf{O} = \mathbf{P}\mathbf{V}$$

Attn Bwd

$$\mathbf{dV} = \mathbf{P}^\top \mathbf{dO}$$

$$\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top$$

$$\mathbf{dS}_i = \mathrm{dsoftmax}(\mathbf{dP}_i) = \left(\mathrm{diag}(\mathbf{P}_i) - \mathbf{P}_i\mathbf{P}_i^\top\right)\mathbf{dP}_i$$

$$\mathbf{dQ} = \mathbf{dS}\mathbf{K}/\sqrt{d}$$

$$\mathbf{dK} = \mathbf{dS}^\top\mathbf{Q}/\sqrt{d},$$

Attn Bwd Bwd

$S = Q K^T \cdot \text{scale}$ (N_Q, N_K)

$P_{ij} = \text{softmax}_j(S_{ij}) = \exp(S_{ij} - L_i)$ (N_Q, N_K)

$dP_{ij} = dO_{id} V_{jd}$ (N_Q, N_K)

$ddS = \text{scale} \cdot (ddQ K^T + Q ddK^T)$ (N_Q, N_K)

$d_{i1} = O_{id} dO_{id}$ $(N_Q, 1)$

$dd_{i1} = ddS_{ik} P_{ik}$ $(N_Q, 1)$

$dP' = dO ddV^T - dP \circ dd - ddS \cdot d + dP \circ ddS$ (N_Q, N_K)

$ddP = P \circ (ddS - dd)$ (N_Q, N_K)

$dS'_{ij} = \text{scale} \cdot P_{ij}(dP'_{ij} - \sum_k dP'_{ik} P_{ik})$ (N_Q, N_K) → b_i

$dS = \text{scale} \cdot P \circ (dP - d)$ (N_Q, N_K)

$dQ' = dS\, ddK + dS' K$ (N_Q, N_D)

$dK' = dS^T ddQ + dS'^T Q$ (N_K, N_D)

$dV' = ddP^T dO$ (N_K, N_D)

$ddO = P\, ddV + ddP V$ (N_Q, N_D)

returns dQ', dK', dV', ddO

inputs $Q, K, V, O, dO, ddQ, ddK, ddV, L$

Multi-Pass Reductions

- 4 total passes in one kernel
- Reduction over Q vs K
- Must avoid saving N x N!
- Linear memory cost

$$S = Q K^T \cdot \text{scale} \qquad (N_Q, N_K)$$

$$P_{ij} = \text{softmax}_j(S_{ij}) = \exp(S_{ij} - L_i) \qquad (N_Q, N_K)$$

$$dP_{ij} = dO_{id} V_{jd} \qquad (N_Q, N_K)$$

$$ddS = \text{scale} \cdot (ddQ K^T + Q ddK^T) \qquad (N_Q, N_K)$$

$$d_{i1} = O_{id} dO_{id} \qquad (N_Q, 1)$$

$$dd_{i1} = ddS_{ik} P_{ik} \qquad (N_Q, 1)$$

$$dP' = dO ddV^T - dP \odot dd - ddS \cdot d + dP \odot ddS \qquad (N_Q, N_K)$$

$$ddP = P \odot (ddS - dd) \qquad (N_Q, N_K)$$

$$dS'_{ij} = \text{scale} \cdot P_{ij}(dP'_{ij} - \underbrace{\sum_k dP'_{ik} P_{ik}}_{b_i}) \qquad (N_Q, N_K)$$

$$dS = \text{scale} \cdot P \odot (dP - d) \qquad (N_Q, N_K)$$

$$dQ' = dS\, ddK + dS' K \qquad (N_Q, N_D)$$

$$dK' = dS^T ddQ + dS'^T Q \qquad (N_K, N_D)$$

$$dV' = ddP^T dO \qquad (N_K, N_D)$$

$$ddO = P\, ddV + ddP V \qquad (N_Q, N_D)$$

returns dQ', dK', dV', ddO

inputs $Q, K, V, O, dO, ddQ, ddK, ddV, L$

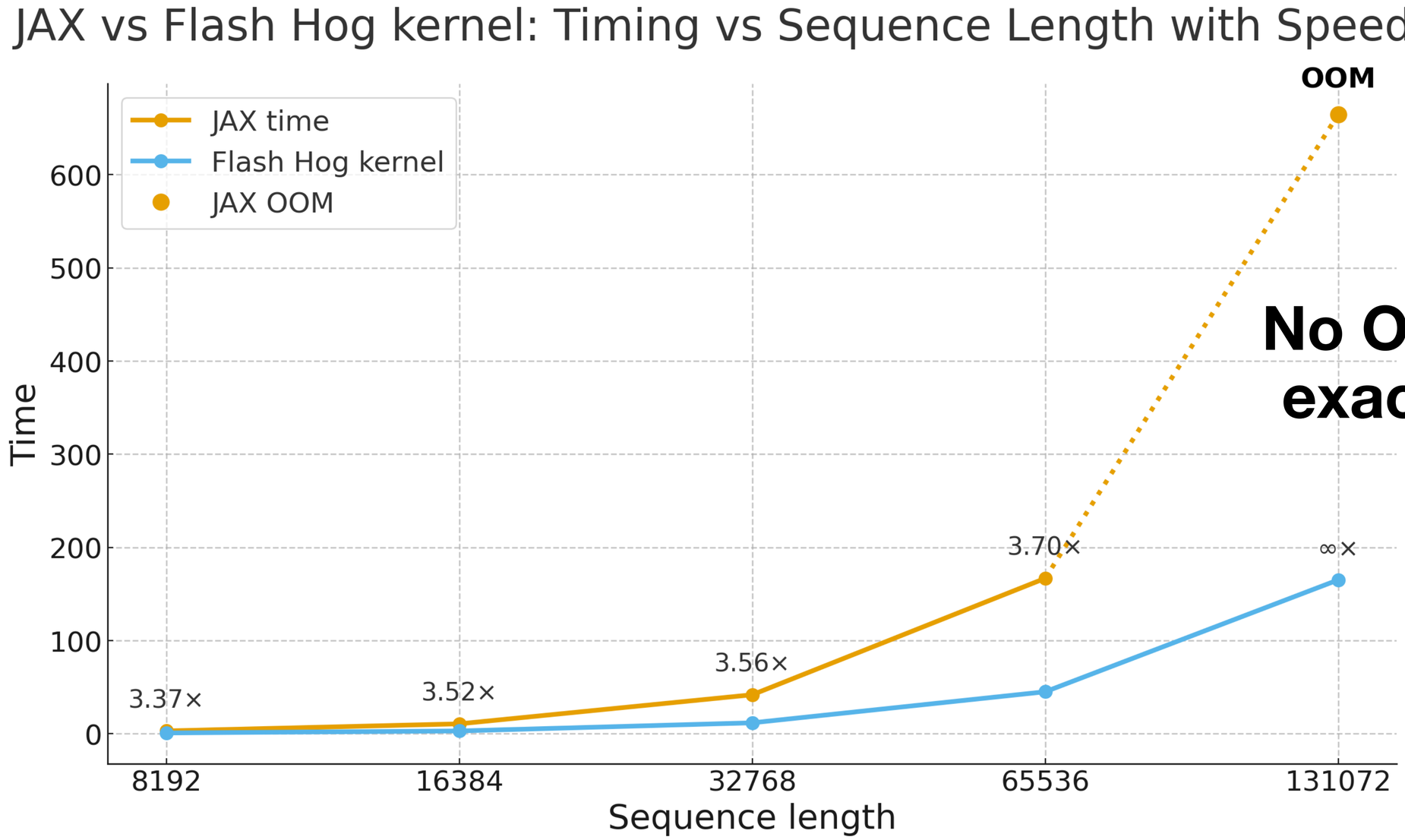

```python
for i in range(T_q):  # COMPUTING dK2_j and dV2_j
    Q_i = tl.load(Q_block_ptr)
    L_i = tl.load(L_block_ptr)
    dO_i = tl.load(dO_block_ptr)
    ddQ_i = tl.load(ddQ_block_ptr)
    dD_i = tl.load(dD_block_ptr)
    B_i = tl.load(B_block_ptr)
    D_i = tl.load(D_block_ptr)
    dD_i = tl.load(dD_block_ptr)

    # Compute attention scores
    S_ij = tl.dot(Q_i, K_j.T) * scale
    P_ij = tl.exp(S_ij - L_i[:, None])

    dP_ij = tl.dot(dO_i, V_j.T)

    ddS_ij = (tl.dot(ddQ_i, K_j.T) + tl.dot(Q_i, ddK_j.T)) * scale

    dP2_ij = (
        tl.dot(dO_i, ddV_j.T)
        - dP_ij * dD_i[:, None]
        - ddS_ij * D_i[:, None]
        + dP_ij * ddS_ij
    )

    dS2_ij = P_ij * (dP2_ij - B_i[:, None]) * scale

    dS_ij = scale * P_ij * (dP_ij - D_i[:, None])

    ddP_ij = P_ij * (ddS_ij - dD_i[:, None])
    dV2_j_acc = tl.dot(ddP_ij.T.to(dO_i.dtype), dO_i, acc=dV2_j_acc)

    dK2_j_acc = tl.dot(dS_ij.T.to(ddQ_i.dtype), ddQ_i, acc=dK2_j_acc)
    dK2_j_acc = tl.dot(dS2_ij.T.to(Q_i.dtype), Q_i, acc=dK2_j_acc)

    # TODO: Try tl.dot((b, 1, k), (b, k, 1)) with acc
    Q_block_ptr = Q_block_ptr.advance((Q_BLOCK_SIZE, 0))
    L_block_ptr = L_block_ptr.advance((Q_BLOCK_SIZE,))
    dO_block_ptr = dO_block_ptr.advance((Q_BLOCK_SIZE, 0))
    ddQ_block_ptr = ddQ_block_ptr.advance((Q_BLOCK_SIZE, 0))
    B_block_ptr = B_block_ptr.advance((Q_BLOCK_SIZE,))
    D_block_ptr = D_block_ptr.advance((Q_BLOCK_SIZE,))
    dD_block_ptr = dD_block_ptr.advance((Q_BLOCK_SIZE,))
```

Implementation: numerics look good ✅

diff(Jax ref, Flash-HOG) ≈ diff(Jax ref, Torch ref)

```
Greatest absolute difference: 0.0126953125 at index (0, 49, 55)
```

Results: >3x speedup, no OOMs!



No OOM@1M: exact HOGs!

Flash-HOG

Higher-order gradients at lightspeed!