FlashSinkhorn

IO-Aware Entropic Optimal Transport

ICML 2026 · Oral

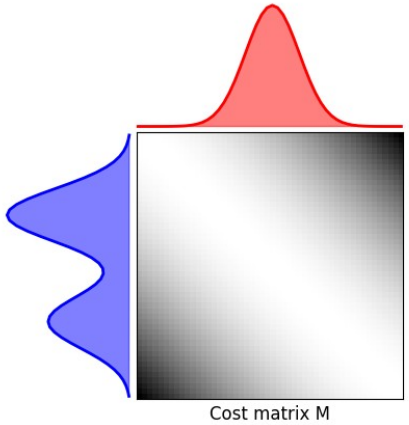
Felix X.-F. Ye[1]† Xingjie Li[2] An Yu[1] Ming-Ching Chang[1] Linsong Chu[3] Davis Wertheimer[3]

[1]University at Albany · [2]UNC Charlotte · [3]IBM Research

`pip install flash-sinkhorn` · arXiv:2602.03067 · github.com/ot-triton-lab/flash-sinkhorn

From optimal transport to Sinkhorn

Optimal transport



Cheapest plan to morph one distribution into another — the **Wasserstein distance**.

Exact OT won't scale

The optimal plan solves a **linear program**: the plan is sparse, found by sequential pivots.

No GPU parallelism — **OOM / times out at scale.**

Fix: entropic regularization

$$P_\varepsilon^* = \arg\min_{P \in \Pi(a,b)} \langle C,\, P \rangle + \varepsilon\, \mathrm{KL}(P \,\|\, a \otimes b)$$

Add an ε·KL entropy penalty ⇒ strictly convex, **dense** plan, fully parallel.

Solved by alternating matrix scaling — **Sinkhorn iterations**.

Sinkhorn makes OT differentiable and GPU-ready — yet every iteration still touches all n × m point pairs.

Sinkhorn: two vectors do all the work

Setup: $C_{ij} = \|x_i - y_j\|^2$ on weighted point clouds (X, a), (Y, b)

Dual potentials f, g (one number per point) encode the plan:

$$P^*_{ij} = a_i\, b_j \exp\left(\frac{f_i + g_j - C_{ij}}{\varepsilon}\right)$$

Choose f, g so the plan hits the marginals; Sinkhorn alternates:

$$f_i \leftarrow -\varepsilon\,\mathrm{LSE}_j\left[\frac{g_j - C_{ij}}{\varepsilon} + \log b_j\right], \qquad g_j \leftarrow -\varepsilon\,\mathrm{LSE}_i\left[\frac{f_i - C_{ij}}{\varepsilon} + \log a_i\right]$$

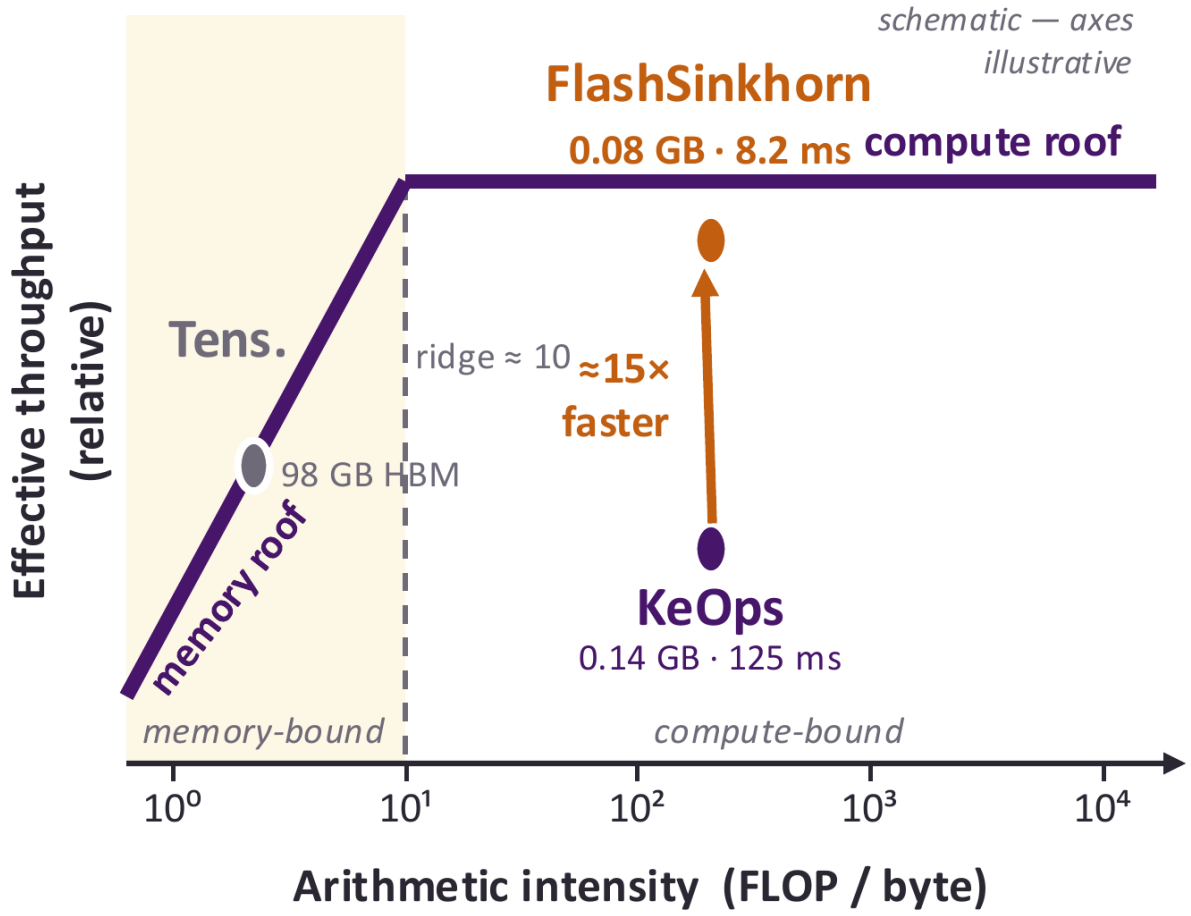
The only state: $f \in \mathbb{R}^n$, $g \in \mathbb{R}^m$ **just two vectors.**

But each update reads every C_{ij}, and C is $n \times m$:

store it ($O(n^2)$ memory) or recompute all pairs on the fly, every iteration

How fast can this inner loop run on a GPU?

Two GPU regimes, both stuck



Store the cost → Tensorized (GeomLoss)

Materializes n×m every iteration. Memory-bound: 79% of cycles stalled, 98 GB HBM traffic.

Recompute on the fly → Online (KeOps, OTT-JAX)

Avoids the matrix, but 854 launches per solve, 9% occupancy — CUDA cores only.

≈15× gap to the compute roof — neither is IO-aware

Already at n=40k, d=512: tensorized **OOMs**, KeOps **times out (>10 min)**.

The unlock: a Sinkhorn step *is* an attention step

$$f_i \leftarrow -\varepsilon \operatorname{LSE}_j\left[\frac{g_j - \|x_i - y_j\|^2}{\varepsilon} + \log b_j\right]$$

expand $\|x_i - y_j\|^2 = \|x_i\|^2 + \|y_j\|^2 - 2x_i^\top y_j$ and set $Q = \sqrt{2}\,X, \quad K = \sqrt{2}\,Y$

absorb: $\hat{g} := g - \|y\|^2 \cdot \hat{f} := f - \|x\|^2 \cdot \delta := \varepsilon \log b$

$$\hat{f} \leftarrow -\varepsilon \operatorname{LSE}_{\text{row}}(S_X), \qquad S_X = \tfrac{1}{\varepsilon}\left(QK^\top + 1_n(\hat{g}+\delta)^\top\right)$$

Row-wise LSE over a **biased $QK^\top$ score**: exactly attention's normalization (Prop. 1)

The dictionary: so we can borrow FlashAttention

Entropic OT	Attention
Source X	$Q = \sqrt{2}\,X$
Target Y	$K = \sqrt{2}\,Y,\quad V = Y$
Regularization ε	Scaling $\sqrt{d}$
$\hat{g} + \delta$	**Key bias**
Barycentric projection	$T_\varepsilon(X) = \mathrm{Softmax}(S_X^*)\,Y$

Why it matters: each Sinkhorn dual update is a **row-wise LogSumExp** over a biased $\mathbf{Q^\intercal K}$ score — attention's exact softmax. So the **n×m score is never stored.**

FlashAttention computes this exact normalization with **far fewer HBM accesses**:

- Tile Q, K through on-chip SRAM
- Keep running max / sum-exp statistics (online LSE)
- Write only the reduced output: the score matrix never exists in HBM

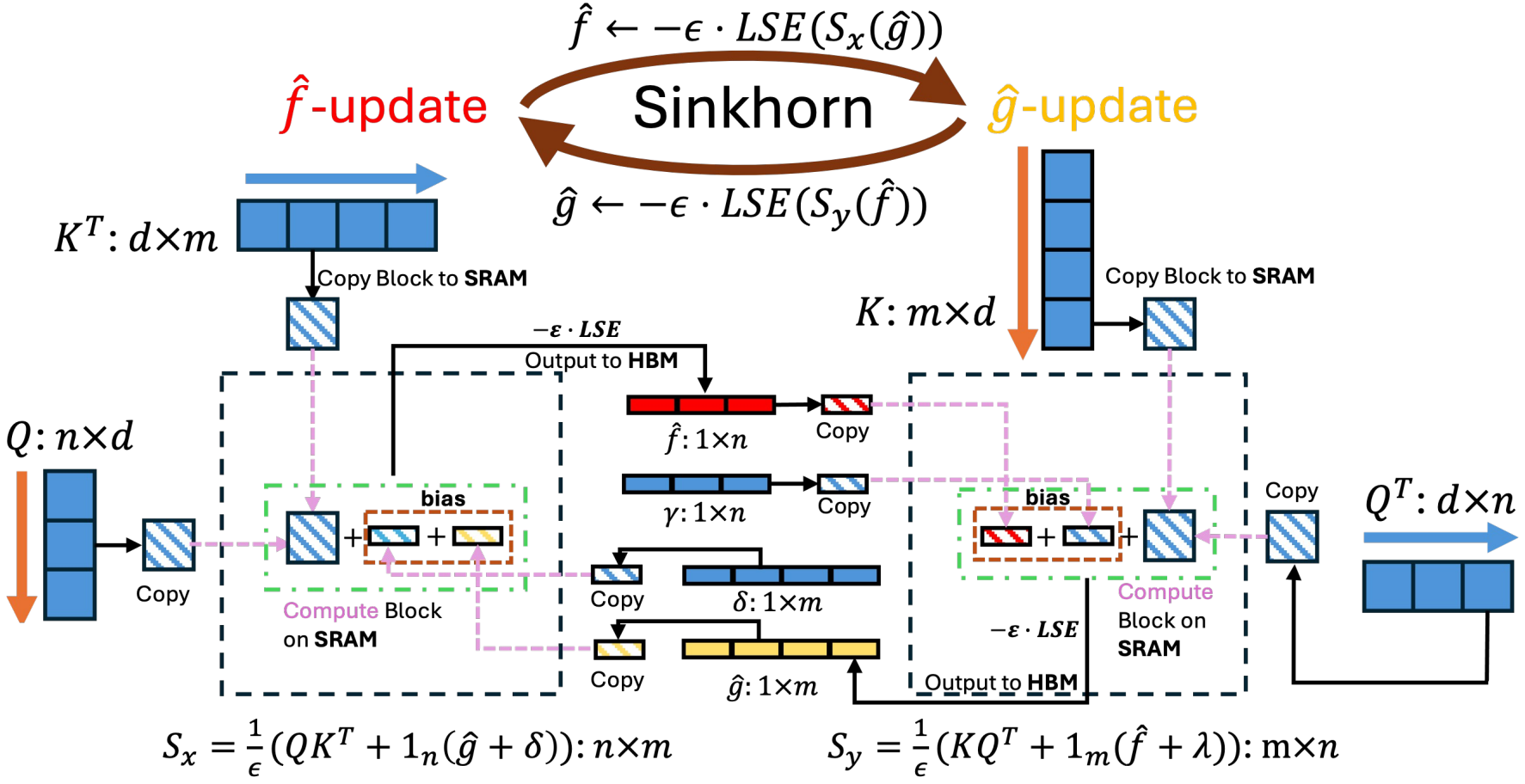
Same trick now legal for Sinkhorn:

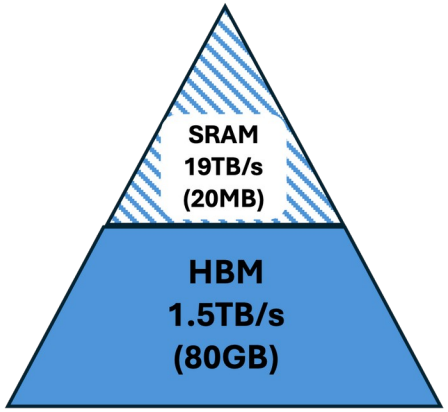
stream tiles, update duals, $O((n+m)\,d)$ **memory**

Works for any cost of the form $C_{ij} = \alpha_i + \beta_j - Q_i^\top K_j$

✓ squared Euclidean · cosine · OTDD × raw $\|x - y\|_2$, neural costs

Tile in SRAM, stream everything





SRAM: 19 TB/s, 20 MB

HBM: 1.5 TB/s, 80 GB

Outer loop holds a Q row-block on-chip; K tiles + bias stream by; one write per row block.

One fused Triton kernel per update: cost, score, LSE, dual write in a single pass.

One streaming kernel: three jobs

Forward solve
online LSE reduction

Analytic gradient
one streamed plan apply

Matrix-free HVP
20–40 streamed plan applies

The shared primitive: streamed plan apply, P never materialized

$$PV = \mathrm{diag}(\mathbf{r})\,\mathrm{Softmax}(S_X(\hat{g}))\,V$$

Gradient: one plan apply against Y, no unrolling

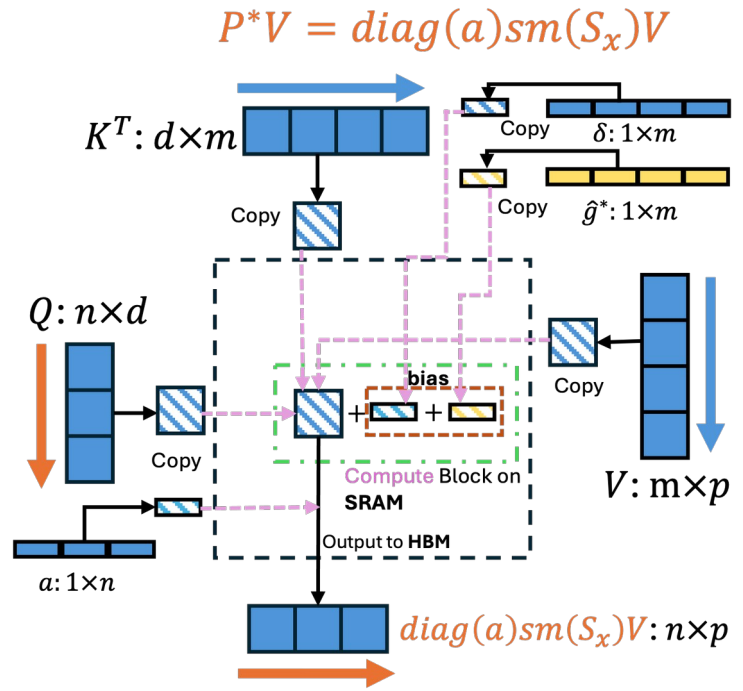
$$\nabla_X \mathrm{OT}_\varepsilon = 2\,\mathrm{diag}(a)(X - \mathrm{Softmax}(S_X^*)\,Y)$$

a = source marginal; S_X^* = score at convergence

HVP: CG on the dual Hessian, every product streamed

$$\mathcal{T}\cdot A = \frac{1}{\varepsilon}\mathcal{R}^\top z + \mathcal{E}\cdot A, \qquad H^* z = \mathcal{R}A \quad (\mathrm{CG})$$

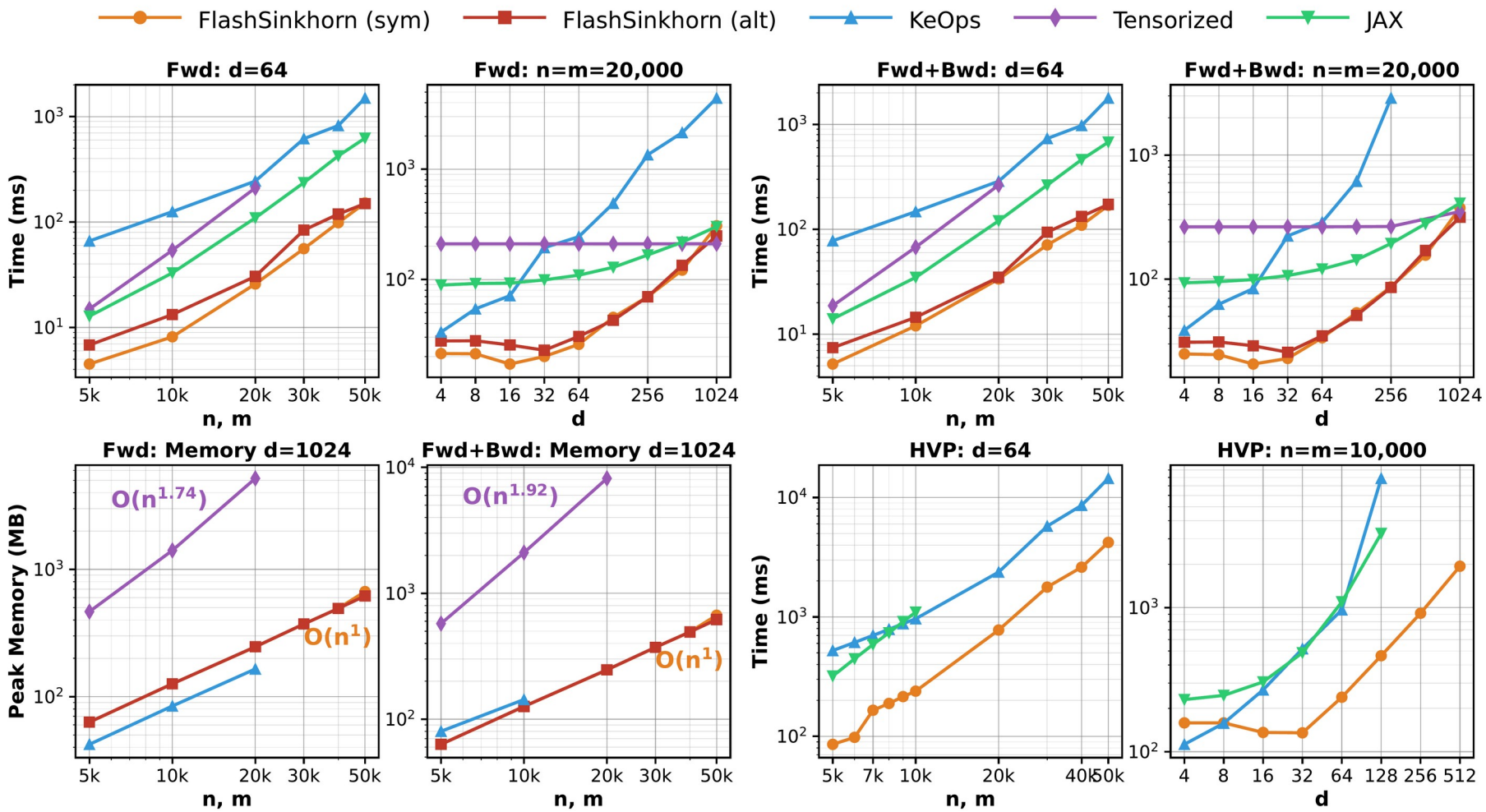
$\mathcal{T} = \nabla_X^2 \mathrm{OT}_\varepsilon$; H^* = dual Hessian; $\mathcal{R}, \mathcal{E}$ = streamed (20–40 plan products/HVP)



Fused matmul → softmax → matmul in SRAM tiles

All three run in $O((n+m)\,d)$ **memory**: first- and second-order OT scale together

How much faster



161×

end-to-end (d=512)

31.7×

forward vs KeOps

17.7×

HVP vs KeOps

Why faster: fusion → 15× less wall-clock at equal FLOPs (8.2 vs 125.5 ms; NCU receipts in backup) · ε = 0.1

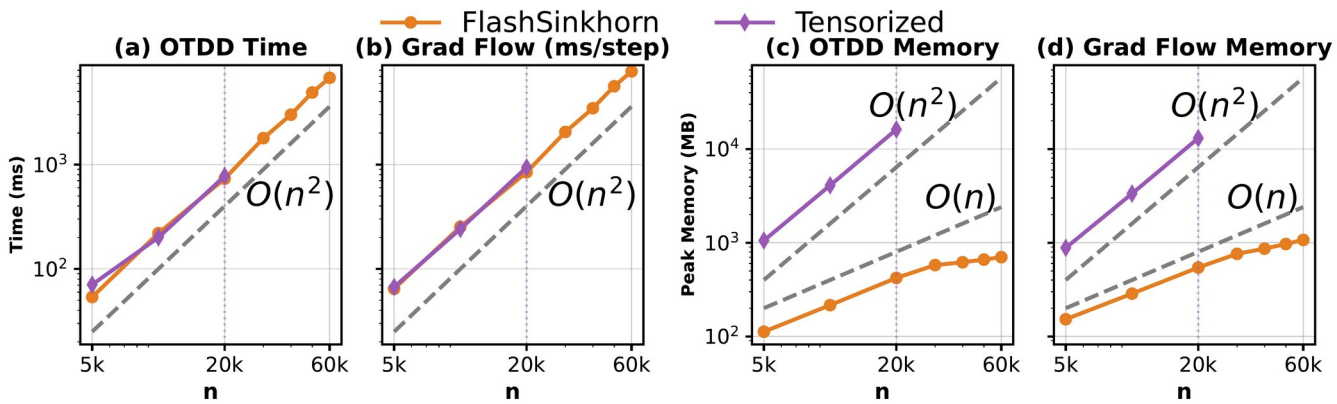
fastest across all n, d tested

O(n) memory: runs where tensorized OOMs

Two applications — first- and second-order, at scale

1 · Dataset distances (OTDD) at scale

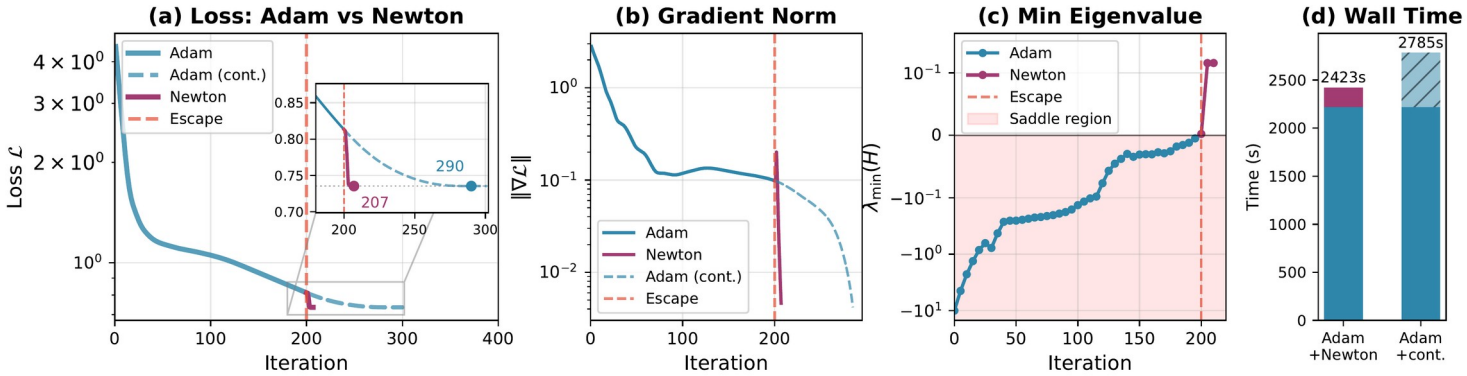
MNIST ↔ Fashion-MNIST, ResNet18 embeddings, d=512



Tensorized baselines OOM beyond n=20k; FlashSinkhorn stays under 1 GB at n=60k.

2 · HVP-driven saddle escape

flow cytometry, n=40k cells, 25 params, 72 random inits



More applications & full results → Poster, Hall A · Thu Jul 9, 5:00–6:45 pm

Takeaways

1. **A Sinkhorn update is an attention step**: row-wise LSE over a biased $QK^\top$ score

2. **FlashAttention-style fusion** → $O((n+m)\,d)$ memory, **up to 161× end-to-end**

3. **One streaming kernel** powers forward, gradient & HVP: **second-order OT scales**

```
loss = SamplesLoss('sinkhorn', blur=0.05)(x, y);  loss.backward()
# streaming forward + backward · drop-in GeomLoss replacement · O(nd), no n×m
```

pip install flash-sinkhorn

arXiv:2602.03067 · github.com/ot-triton-lab/flash-sinkhorn · † xye2@albany.edu

Thank you! **Come by the poster → Hall A · Thu Jul 9 · 5:00–6:45 pm**

Why faster: NCU receipts

Forward · n=m=10k, d=64

n=m=10k	Ten	KeO	Flash
Runtime ms	54.0	125.5	**8.2**
HBM GB	**98**	0.14	**0.08**
SM util %	98	49	74
Mem stall %	**79**	8	**3**
Occupancy %	89	**9**	**11**
Instr (B)	10	**16**	**7**
Launches	—	**854**	**130**
Tensor-pipe (M)	—	**3.5**	**10.1**
Bottleneck	**Mem**	Comp	**Comp**

Root causes · NCU counters

Tensorized

SMs active but **stalled on the n×m matrix** (98% util misleading)

KeOps

854 launches vs 130 fused: dispatch dominates

FlashSinkhorn

2.9× more tensor-core compute via tiled dot-products

Fusion turns the same FLOPs into **15.3× less wall-clock** (8.2 vs 125.5 ms).

Instr (B) = instructions executed, billions.

Backup · per-kernel anatomy of the forward pass

KeOps: 96 GpuConv1D reductions + 758 elementwise auxiliaries = **854 launches** per evaluation

FlashSinkhorn: ~5 fused Triton kernels per iteration = **130 launches** (6.6× fewer)

Tensor-pipe instructions: 10.1M vs 3.5M: **2.9× more compute through tensor cores** via tiled tl.dot

Registers: 255/thread (GPU max); fusion trades occupancy (11%) for far less memory traffic (0.08 GB vs 98 GB)

Backup · HVP decomposition & indefiniteness

$$\mathcal{T} = \frac{1}{\varepsilon}\mathcal{R}^\top (H^*)^\dagger \mathcal{R} + \mathcal{E}$$

implicit term: PSD (H^* is the dual Sinkhorn Hessian) · explicit term $\mathcal{E}$: indefinite

$$\mathcal{E}_{k,:,k,:} = 2a_k I_d - \frac{4}{\varepsilon}\sum_j P^*_{kj}(x_k - y_j)^\top (x_k - y_j)$$

Positive diagonal – scaled displacement covariance: large displacement vs ε ⇒ negative eigenvalues

That indefiniteness is what the saddle-escape application exploits (λ_{min} monitoring).

Backup · when does the trick apply?

$$C_{ij} = \alpha_i + \beta_j - Q_i^\top K_j$$

Any cost with precomputable per-point terms + an explicit feature dot-product streams the same way:

- ✓ squared Euclidean (Q = √2 X, K = √2 Y): the primary instance
- ✓ cosine distance (after L2 normalization)
- ✓ OTDD label-augmented cost (bounded additive lookup, evaluated in-kernel)
- ✗ raw Euclidean ‖x−y‖ (square root breaks the affine form), learned neural costs: future work

Backup · limitations & open directions

Limitations

- ▸ fast path = squared Euclidean cost only
- ▸ small n (< 5k): launch overhead can negate gains
- ▸ extreme aspect ratios ($n \ll m$) underutilize row-parallelism; FA-2-style partitioning is the fix
- ▸ autotuning: one-time warmup per problem shape

Open directions

- ▸ general / learned cost functions
- ▸ LLM alignment: OT over response distributions in training loops
- ▸ curvature-aware optimization at scale (trust-region Newton, Lanczos)
- ▸ block-sparse multiscale for n > 100k

Backup · full speedup tables

Forward & end-to-end (× vs baselines)

		Forward ×			Fwd+Bwd ×		
n	d	KeO	Ten	JAX	KeO	Ten	JAX
10k	128	9.4	3.2	2.0	10.3	3.5	2.0
10k	512	**31.7**	1.7	1.3	**161**	1.7	1.2
40k	128	12.5	**OOM**	3.0	13.5	**OOM**	2.9
40k	512	**OOT**	**OOM**	1.6	**OOT**	**OOM**	1.6

HVP, second-order (×)

n	d	KeO	JAX
10k	64	4.0	4.6
10k	128	**17.0**	7.0
8k	128	**17.7**	6.0
6k	256	**OOT**	**12.5**

OOM / OOT = out of memory / out of time (10 min) · A100-80GB, ε = 0.1

Backup · where the LSE update comes from

KKT conditions give the optimal plan through dual potentials f, g:

$$P^*_{ij} = a_i\, b_j \exp\left(\frac{f_i + g_j - C_{ij}}{\varepsilon}\right)$$

1. Enforce the row marginal and take logs:

$$\sum_j P^*_{ij} = a_i \quad \Longrightarrow \quad \frac{f_i}{\varepsilon} + \log \sum_j \exp\left(\frac{g_j - C_{ij}}{\varepsilon} + \log b_j\right) = 0$$

2. Solve for f: the update is a pure row-wise LogSumExp reduction:

$$f_i \leftarrow \; - \varepsilon\, \mathrm{LSE}_j\left[\frac{g_j - \|x_i - y_j\|^2}{\varepsilon} + \log b_j\right]$$

Alternate f and g until the marginals match: **that is Sinkhorn**