DRAINSINKHORN: INSTANCE-RESIDENT BATCHED ENTROPIC OPTIMAL TRANSPORT ON GPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

Many learning pipelines call entropic optimal transport (OT) inside a loop, and each call is a batch of small, independent problems: one per image in detection label assignment, per patch pair in point-cloud registration, per frame pair in tracking, and per tile in domain decomposition. General GPU solvers launch every Sinkhorn update over the whole batch and run the batch until its slowest problem stops, so practitioners write a kernel for each shape or drop OT. We present DRAINSINKHORN, a batched Sinkhorn solver that makes each problem of the batch, an instance, the unit of GPU execution. Wherever an instance fits on chip, one GPU thread block solves it from start to finish and exits as soon as the instance converges, freeing its slot for a pending instance; with a fixed iteration count, several instances share a thread block, and the forward map and its exact gradient take one launch each. Larger instances leave later launches once they converge. On tracking and domain-decomposition workloads, DRAINSINKHORN is 2.73–10.75× faster than OTT-JAX, the fastest general batched GPU solver in our comparison, and 5.3–34× faster than POT; the gain is largest on calls with thousands of instances. In training layers it is 1.05–2.8× faster than OTT-JAX and 9.9–24× faster than the Sinkhorn modules of OTA and GeoTransformer, cutting label assignment from 16.3% to 4.7% of each GPU's training step and making GeoTransformer's patch matching 10.4× faster in a training step.

1 INTRODUCTION

Many pipelines call entropic optimal transport (OT) inside a loop, and each call is a batch of small, independent problems. At inference, a multi-object tracker couples the detections of consecutive frames, domain decomposition splits a large problem into cell problems that are "numerous, small, and with diverse shapes" (Medina & Schmitzer, 2025; Bonafini & Schmitzer, 2021), and coarse-to-fine registration solves one problem per matched patch pair (Yu et al., 2021; Qin et al., 2022). In training, the Sinkhorn iterations are themselves a layer: manifold-constrained hyper-connections (mHC) project one 4×4 residual mixing matrix per token and layer with 20 iterations (Xie et al., 2025), Optimal Transport Assignment (OTA) assigns detection labels per image with 50 (Ge et al., 2021a), and GeoTransformer matches 256 patch pairs of 65×65 points per 3DMatch scan pair with 100 (Qin et al., 2022). Each step thus produces B problems, each with its own cost, marginals, and regularization, that must all be solved before the step continues. Similar scalability pressures arise in other large-scale learning pipelines; in graph-based recommendation, linear-time GNNs bring training cost down to that of matrix factorization (Zhang et al., 2024).

These calls are expensive enough that practitioners work around them. They write kernels for one shape, such as the fused $n = 4$ Sinkhorn kernel of mHC (Xie et al., 2025), the warp-level 4×4 solver of Wang & Qiu (2026), and the CUDA cell solver of Medina & Schmitzer (2025), or they drop OT: YOLOX replaces OTA's Sinkhorn by a top-k rule because Sinkhorn adds 25% training time (Ge et al., 2021b), and LightGlue replaces SuperGlue's Sinkhorn as expensive in compute and memory (Lindenberger et al., 2023). We measured the same effect: with a COCO-trained detector on an A100, OTA's label assignment takes 16.3% of a training step on each GPU, 12.8% in its Sinkhorn loop (section 5.2). General GPU solvers organize the work around the batch (Flamary et al., 2021; Cuturi et al., 2022; Feydy et al., 2019): each half-step, one of the two Sinkhorn updates, is a launch over all problems, the batch runs until its slowest problem stops, and unrolled differentiation, where used, records every iteration. For many small problems this organization wastes time in two ways.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

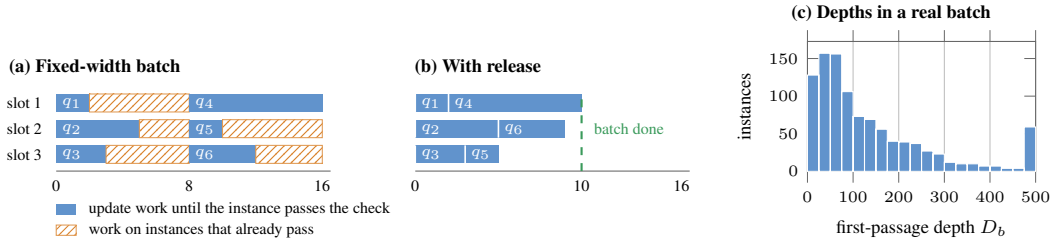


Figure 1: **Instances finish at different depths; fixed-width execution pays for the slowest one.** (a) Six instances with depths $(2, 5, 3, 8, 2, 4)$ on three slots; a fixed-width batch runs every instance as long as the slowest one in the batch (hatched). (b) With release, a finished instance frees its slot at once for a pending one: 10 instead of 16 time units (illustration). (c) First-passage depths, the iterations each instance needs to reach the tolerance, of 1024 MOT17 frame pairs (`mot-b16` pool of figure 3c); the last bin holds instances at the cap of 500.

One update does little arithmetic compared with launching it and synchronizing with the host, and problems need very different numbers of iterations: in the MOT17 association workload of section 5, the iterations needed to reach a two-sided tolerance of 10^{-4} range from 5 to the cap of 500 with a median of 85, so 73% of the updates in a batch run to its slowest problem go to problems that already pass (figure 1).

DRAINSINKHORN instead makes each problem of the batch, which we call an *instance*, the unit of GPU execution. Wherever an instance fits on chip, its whole state stays in the registers of one GPU program (a thread block) from the first update to the last; we call such an instance *resident*. When a resident instance converges, its program exits and the GPU starts a pending instance in the freed slot; we call this *release*. We make three contributions.

- **Instance-resident execution for both stopping rules** (section 3): tolerance solves that release each instance by program exit, for dense instances of up to 64 points per side, point clouds in $d \leq 4$ with up to 512, and grids; fixed-iteration solves that pack several instances into one program and run the forward map and its exact backward in one launch each, without host synchronization; and instances with one short and one very long side, in one launch when they fit on the device.
- **Per-instance release with a runtime model** (sections 3 and 4). Instances too large for one program run on an indexed path that removes converged instances from every later launch. A list-scheduling bound explains when release shortens a batch whose programs run in several waves, the groups of programs that the GPU runs at once.
- **An in-loop benchmark** (section 5) built from the users above: label assignment captured from COCO with a trained detector, mHC projections, patch matching, tracking, and domain decomposition. Each is compared with the application's own implementation where one exists and with general solvers under common accuracy gates, and we report the solver's share of a training step.

On label assignment with two images per call, one GPU's share of OTA's 8-GPU step, DRAINSINKHORN is 13.0× faster than OTA's own implementation and 6.31× faster than a padded batch of the same updates. It cuts assignment from 16.3% to 4.7% of a detector's training step without changing any assignment, and makes GeoTransformer's patch matching 10.4× faster in a training step (table 1). On the inference workloads it is the fastest of all compared methods, 2.73–10.75× faster than the fastest baseline, OTT-JAX, and 5.3–34× faster than POT (table 2). Most of the gain comes from on-chip execution; release adds 1.10–1.60× on multi-wave batches (section 5.4).

2 PROBLEM

Batches of independent problems. A batch is a set $\{q_b\}_{b=1}^{B}$ of balanced entropic OT problems $q_b = (C_b, a_b, b_b, \varepsilon_b)$ with a cost $C_b \in \mathbb{R}^{N\times M}$, given either densely or as $C_{b,ij} = \|x_{b,i} - y_{b,j}\|^2$ for supports $X_b \in \mathbb{R}^{N\times d}$, $Y_b \in \mathbb{R}^{M\times d}$, and nonnegative marginals of equal mass. We call each q_b an

instance of the batch. Instance b is

$$\min_{P \geq 0} \langle C_b, P \rangle + \varepsilon_b \sum_{ij} P_{ij}(\log P_{ij} - 1) \quad \text{s.t.} \quad P\mathbf{1} = a_b, \quad P^\top \mathbf{1} = b_b, \tag{1}$$

and its solution P_b is the plan of instance b. Instances share no variables, and B counts instances, not points. Instances may share a support, as domain-decomposition tiles do, but not solver state.

Updates. We use log-domain Sinkhorn with dual potentials (f, g) and the plan $P_{ij} = \exp((f_i + g_j - C_{ij})/\varepsilon)$. One iteration is the row update followed by the column update; we call each of the two a *half-step*:

$$f_i \leftarrow \varepsilon\big(\log a_i - \mathrm{LSE}_j\big((g_j - C_{ij})/\varepsilon\big)\big), \qquad g_j \leftarrow \varepsilon\big(\log b_j - \mathrm{LSE}_i\big((f_i - C_{ij})/\varepsilon\big)\big), \tag{2}$$

starting from $f = g = 0$. After the column update, $P^\top \mathbf{1} = b$ exactly; the row marginal does not.

Stopping at a tolerance. Let $r_b^{(t)} = \max\{\|P\mathbf{1} - a_b\|_1, \|P^\top \mathbf{1} - b_b\|_1\}$ be the two-sided residual of instance b after t iterations. Checks happen every q iterations; instance b stops at the first check with $r_b^{(t)} \leq \tau$ and returns that iterate. We call this number of iterations its *first-passage depth* D_b. An instance that has not passed after the cap of $K_{\max}$ iterations returns its last iterate with status *capped*.

Running exactly K iterations. When Sinkhorn is a layer of a trained model, the layer is the K-step map itself: mHC projects each residual mixing matrix with 20 iterations (Xie et al., 2025), OTA assigns labels with 50 (Ge et al., 2021a), and GeoTransformer matches patches with 100 (Qin et al., 2022). With this stopping rule the solver runs exactly K iterations of (2) from $f = g = 0$ and returns that iterate. When gradients are requested, it also returns the exact vector–Jacobian product of the K-step map with respect to C_b, a_b, b_b, and ε_b, obtained by unrolled differentiation rather than as an implicit gradient at the optimum. Under either rule, for each instance in input order, the solver returns (f_b, g_b) or the plan P_b, together with the residuals, D_b, and the status.

What fixed-width execution costs. A fixed-width batch runs every instance until the slowest one stops. It performs $BD_{\max}$ instance-updates, one per iteration of one instance, where $D_{\max} = \max_b D_b$, while stopping each instance at its own depth needs only $\sum_b D_b$. The ratio

$$R_{\mathrm{work}} = \frac{B\, D_{\max}}{\sum_b D_b} \geq 1 \tag{3}$$

depends only on the workload; across our workloads it ranges over 1.19–4.70. Section 4 relates it to runtime.

3 INSTANCE-RESIDENT EXECUTION

DRAINSINKHORN makes the instance, not the batch, the unit of GPU execution. Wherever an instance fits, one GPU program, Triton's name for a thread block, runs the instance's whole solve: the instance's state stays on chip from the first update to the last, and the host issues one launch for the batch. Each program reads and writes only its own instance's data. The execution path depends only on the stopping rule and the instance shape (figure 2). With a tolerance, dense instances with $N, M \leq 64$ and squared-Euclidean instances with $d \leq 4$ and $N, M \leq 512$ run as resident solves with release (section 3.1), and regular grids with at most 16 points per axis can request a separable resident solve (section 3.2). With a fixed iteration count, dense instances with $N, M \leq 128$ run as packed resident forward and backward passes (section 3.3), and short-by-long instances, such as the images of a label-assignment step, run in chunks along the long side (section 3.4). All other instances use indexed execution over unfinished instances (section 3.5). All paths return one per-instance result type, and all kernels are written in Triton.

3.1 RESIDENT SOLVES WITH RELEASE

When both supports have at most 64 points, the cost tile of one instance, its marginals, and its potentials fit in the registers of one program. DRAINSINKHORN assigns one program to each instance

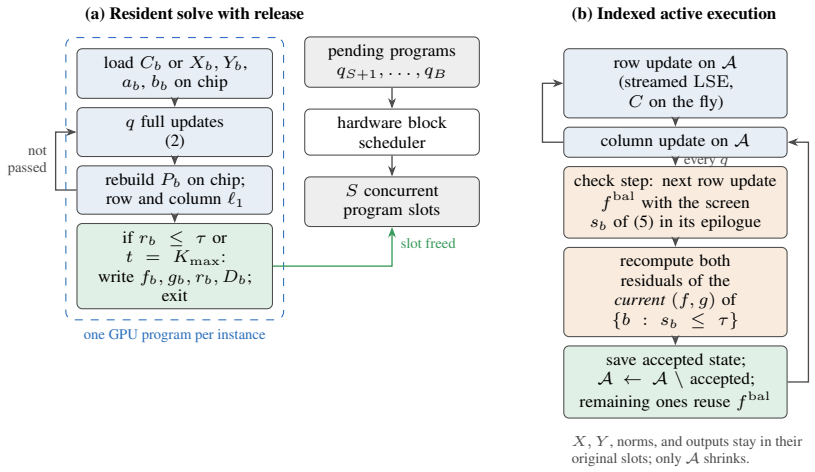


Figure 2: **Two realizations of per-instance release.** (a) Instances that fit on chip run the whole loop in one program; release ends the program and the hardware scheduler fills the slot. The fixed-iteration path (section 3.3) runs the same program without the check, several instances per program. (b) Larger instances iterate over an index set $\mathcal{A}$; instances that pass the screen are checked on their current iterate, and those that pass leave $\mathcal{A}$.

and runs the complete solve there. Each program repeats: perform q iterations of (2) on chip; rebuild the plan tile from the current potentials; reduce the tile's rows and columns to the two marginal ℓ_1 residuals; stop if both are at most τ or the cap is reached. On release the program writes f_b, g_b, both residuals, and D_b, and exits.

Release is program termination. The batch is one launch of B programs, and the GPU block scheduler starts a pending program whenever a running one exits, so release needs no host decision, index vector, or synchronization until the batch is done.

Larger low-dimensional supports. For squared-Euclidean costs with $d \le 4$ and up to 512 points per side, the cost tile no longer fits, but the coordinates, squared norms, marginals, and potentials of one instance do. The program keeps these on chip and recomputes distances tile by tile inside each half-step, so neither C nor P is materialized; the check and release are unchanged. Whether this path or the indexed one is faster depends on the device. On first use, DRAINSINKHORN therefore runs the indexed path and three resident tilings on the calling batch, admits a tiling only if its registers do not spill to memory and its per-instance stopping status matches the indexed path's, and keeps the fastest tiling per device, batch size, and shape for the rest of the process.

3.2 SEPARABLE GRIDS

Regular grids, such as the cells of a domain decomposition, have Cartesian supports, and the squared Euclidean cost splits as $C_{(i_1 i_2)(j_1 j_2)} = C^{(1)}_{i_1 j_1} + C^{(2)}_{i_2 j_2}$. The log-sum-exp of a half-step then factorizes into two one-axis reductions,

$$\mathrm{LSE}_{j_1, j_2}\left(h_{j_1 j_2} - C^{(1)}_{i_1 j_1}/\varepsilon - C^{(2)}_{i_2 j_2}/\varepsilon\right) = \mathrm{LSE}_{j_1}\left(-C^{(1)}_{i_1 j_1}/\varepsilon + \mathrm{LSE}_{j_2}\left(h_{j_1 j_2} - C^{(2)}_{i_2 j_2}/\varepsilon\right)\right), \quad (4)$$

which lowers the work of a half-step on an $s \times s$ grid from s^4 to $2s^3$. GPU domain-decomposition solvers use this reduction (OT Group Göttingen, 2025; Medina & Schmitzer, 2025); DRAINSINKHORN runs it inside a resident program. With at most 16 points per axis, the axis costs, the potentials, and the intermediate $s \times s$ array fit in one program, so the resident loop above, including the check and release, runs unchanged on grids of up to 256 points per side. The option is explicit: the caller supplies the grid shapes.

3.3 FIXED-ITERATION FORWARD AND BACKWARD

A training-time layer calls the solver in every step, often on very many tiny instances (mHC projects one 4×4 matrix per token and layer), and needs gradients. There is nothing to release, but the costs of batch-organized execution remain: one launch per half-step, and an autograd graph that stores the intermediate plans of all K iterations. DRAINSINKHORN runs the whole K-step map of an instance in one resident launch and its vector–Jacobian product in one more.

Packing. A 4×4 instance would leave almost all threads of a program idle. One program therefore holds several instances along a leading axis: 32 for padded tiles of at most 64 elements (the mHC shapes) and $\max(1, \lfloor 2048/(RC) \rfloor)$ for larger tiles of $R \times C$ elements, with masks for padding and for the partly filled last program. Reductions run only along each instance's own rows and columns, so packing changes no instance's iterates.

Forward. Each program loads the cost tiles, marginals, and ε_b of its instances and runs the K iterations on chip in base 2, with potentials and costs prescaled by $\log_2 e/\varepsilon_b$ so that no half-step divides. The program writes the final potentials and, on request, the plan and both marginal residuals from the same tile. For gradients, it also writes the potential history $(f^{(t)}, g^{(t)})_{t \le K}$, $K(N+M)$ numbers per instance instead of K plans of NM (appendix G).

Backward. A second resident launch reloads the cost tiles, turns the adjoint of the plan into adjoints of C_b, ε_b, and the final potentials, and walks the $2K$ half-steps in reverse. The normalized weights $\exp((g_j - C_{ij})/\varepsilon - \mathrm{LSE})$ of each half-step are rebuilt on chip from the saved potentials and consumed immediately, and the adjoints of f and g accumulate into gradients with respect to C_b, a_b, b_b, and ε_b. The result is the exact vector–Jacobian product of the K-step map.

Host control. This path issues a fixed, data-independent number of launches; without input value checks, the forward and backward passes need no host synchronization and fit in a CUDA graph.

3.4 SHORT-BY-LONG INSTANCES

We call an instance short-by-long when one side has at most 128 points and the other more than 4096. Label assignment is the example: each image couples its few ground-truth boxes with tens of thousands of anchors, the candidate locations of a detector (section 5.1). A whole instance does not fit on chip, but its short side does. DRAINSINKHORN splits the long side into chunks, one program each. A program keeps the short-side potential in registers, updates the long-side potential of its chunk by a reduction over the short side, and leaves one partial log-sum-exp per short index; the next short-side update starts by combining these partials. Programs visit rows in blocks over each instance's actual short side, so images with few boxes do no work on padded rows. When a call needs at most one program per streaming multiprocessor (SM), with chunks of 1024 or 2048 entries, a single cooperative launch, for which the driver guarantees that all programs run at the same time, runs all K iterations with a counter between them; otherwise each iteration is one launch that reads the cost once. Because a box with no anchor inside would carry a potential near OTA's outside-box cost of 10^6, which FP32 cannot resolve, DRAINSINKHORN first subtracts each short-side entry's minimum cost and starts that entry's potential at minus the shift. In exact arithmetic this runs the same K-step map from $f = g = 0$ whichever side is updated first. Costs are stored short side first, and either side may be updated first, so a transposed view reproduces OTA's order.

3.5 INDEXED ACTIVE EXECUTION

Larger instances do not fit on chip, so each half-step is a streamed kernel: for a tile of rows, the kernel computes squared distances from centered coordinates and cached norms, accumulates an online log-sum-exp over column tiles, and writes the updated potential, without materializing C or P (Feydy et al., 2019; Ye et al., 2026). Every launch takes an index vector $\mathcal{A}$ of unfinished instances, and the programs of the launch cover only these $|\mathcal{A}|$ instances; state and outputs keep their batch slots.

Screen in the next update. At a check, the column marginal is exact, so the two-sided residual reduces to the row residual. Let f^{bal} be the next row update computed from the current g. The row

marginal of the current plan is $(P\mathbf{1})_i = a_i \exp((f_i - f_i^{\mathrm{bal}})/\varepsilon)$, hence

$$\|P\mathbf{1} - a\|_1 = \sum_i a_i \left|\mathrm{expm1}\big((f_i - f_i^{\mathrm{bal}})/\varepsilon\big)\right|. \tag{5}$$

The identity is known (Ye et al., 2026); we use it as a screen that costs no extra pass, because the final stage of the row-update kernel that produces f^{bal} also accumulates (5). Instances with screen value at most τ go to a separate streamed kernel that recomputes both marginals of the current iterate; instances that pass copy f, g, and residuals to the output and leave $\mathcal{A}$. For instances that stay active, f^{bal} is exactly their next row update, so the screen adds only that final stage. Each check costs one device-to-host transfer of the pass flags and one upload of the new index vector; at the cap, the remaining instances are checked once more and marked capped (Algorithm 1, appendix B).

4 Work-to-runtime analysis

Release removes work; this section states when it also saves time, in exact arithmetic, with proofs in appendix A. With $\mathcal{A}_t = \{b : D_b \geq t\}$ the set of instances active at iteration t, $\sum_{t \leq D_{\max}} |\mathcal{A}_t| = \sum_b D_b$, so release removes $BD_{\max} - \sum_b D_b$ instance-updates (6).

Resident path: waves of programs. Suppose the GPU runs at most S programs at once, one per slot, the program of instance b takes $\theta D_b + \sigma$ time (θ per iteration and a fixed cost σ), and programs do not slow each other down. The block scheduler is work-conserving: it starts a pending program whenever a slot is free. The programs that run at the same time form a *wave*.

Proposition 1 (Resident makespan)**.** *Under these assumptions the makespan T_{active} of execution with release satisfies*

$$\max\Big(\tfrac{1}{S}\textstyle\sum_b(\theta D_b + \sigma),\ \theta D_{\max} + \sigma\Big) \ \leq\ T_{\mathrm{active}} \ \leq\ \tfrac{1}{S}\textstyle\sum_b(\theta D_b + \sigma) + \big(1 - \tfrac{1}{S}\big)(\theta D_{\max} + \sigma), \tag{7}$$

while a fixed-width batch that runs every program to $D_{\max}$ takes $T_{\mathrm{static}} = \lceil B/S \rceil (\theta D_{\max} + \sigma)$.

The upper bound is Graham's bound for list scheduling (Graham, 1969). With $\sigma = 0$, the ratio $T_{\mathrm{static}}/T_{\mathrm{active}}$ lies in an interval computable from the depths alone, which section 5.4 compares with measurement. When the batch fits in one wave ($B \leq S$), T_{static} equals the lower bound, so release cannot help. When the batch spans many waves, $T_{\mathrm{static}}/T_{\mathrm{active}} \to (\theta D_{\max} + \sigma)/(\theta \bar{D} + \sigma)$ for the mean depth $\bar{D}$, and this limit approaches R_{work} of (3) as $\sigma \to 0$.

Indexed path: widths and control. An indexed half-step over a active instances with n rows launches $a\lceil n/b_m \rceil$ programs for row tile size b_m, which the GPU runs in waves of at most Q programs, so an iteration costs about $g_n(\lceil a\lceil n/b_m \rceil / Q \rceil)$ for a function g_n that increases with the number of waves. Shrinking $\mathcal{A}$ therefore saves time only when it removes a wave, and every check adds host control (section 3.5). With few row tiles per instance, the removed work (6) can be large while the removed time is small; the resident path targets these small instances.

5 Experiments

5.1 Setup

Inference workloads. Seven workloads, each a batch of 64 to 3771 instances with 16 to 1024 points per side (appendix C). *MOT17* (Milan et al., 2016): detections of consecutive frames are matched in a four-dimensional box feature space; one call holds all frame pairs of the sequences (offline association), grouped by detection count into `mot-b16`, `mot-b32`, and `mot-b64`, or cut into windows of 1024 detections (`mot-win`). *Domain decomposition* (DD): images of the DomainDecomposition reference (Bonafini & Schmitzer, 2021) split into 4×4, 8×8, and 16×16 tiles (`dd-t4`, `dd-t8`, `dd-t16`), a controlled proxy for DD cell problems.

Training layers. *OTA label assignment* is captured from real data: 320 COCO val2017 images in 20 steps of 16, with OTA's official cost, supplies, and demands (Ge et al., 2021a) built from a COCO-trained FCOS (Tian et al., 2019) at its training resolution (appendix H). Each step has 16

Table 1: **Training layers, fixed iterations** (other method's time over DRAINSINKHORN's, paired per block; bold: other faster; †fails the output gate; *computes a different quantity: TileLang-mHC adds 10^{-6} to every entry after its first normalization, and its backward is an implicit gradient at its output, not the gradient of the 20-step map). (a) OTA, 20 COCO steps, 50 iterations, 2 or 16 images per call; share: assignment share of the step (median; appendix H), official → DRAINSINKHORN. (b) mHC, 4×4 inputs captured from a trained GPT, 20 iterations, forward / forward+backward, all replayed from CUDA graphs. (c) GeoTransformer, 32 3DMatch pairs, 100 iterations; in model: OT time inside the model's step, median over pairs; alone: forward calls; share: OT share of the step.

(a) Label assignment

Call	images	DRAINSINKHORN (ms)	OTA official	padded	POT	OTT	share (%)
one GPU's share	2	1.88	13.0	6.31	4.59	1.05	16.3 → 4.7
whole step	16	7.95	24.4	6.29	4.85†	2.15	15.4 → 4.4

(b) mHC projections

Input	B	DRAINSINKHORN (ms)	`torch.compile`	triton-sinkhorn	TileLang-mHC*
hard layer	4K	0.057 / 0.244	2.94 / 2.66	7.91 / 2.56	**0.68 / 0.22**†
medium layer	4K	0.057 / 0.246	2.95 / 2.63	7.86 / 2.54	**0.68 / 0.21**†
easy layer	4K	0.057 / 0.243	2.91 / 2.60	7.88 / 2.57	**0.68 / 0.22**†
16 layers	64K	0.151 / 0.453	1.83 / 2.60	41.1 / 14.3	1.02† / **0.48**†

(c) Patch matching

Setting	DRAINSINKHORN (ms)	official	OTT	POT	share (%)
alone, forward	1.34	10.0	2.80	11.9	–
in model, inference / training	1.43 / 2.65	9.94 / 10.4	–	–	17.2 → 2.1 / 16.4 → 1.9

instances of 2 to 54 rows, padded to 27K to 36K anchors; 50 iterations, unnormalized marginals. *mHC*: logits and upstream gradients captured from a trained mHC model, and synthetic logits $\sim \mathcal{N}(0, \sigma^2)$, $\sigma \in \{1, 10\}$ (Wang & Qiu, 2026), with $n \in \{4, 8\}$ and up to 1M instances per call; 20 iterations. *Patches*: GeoTransformer's fine matching captured from its released model on 32 3DMatch test pairs (Qin et al., 2022); each pair gives 256 instances of 64×64 scores plus a dustbin row and column that absorb unmatched points, with a median of 16 valid points per side; 100 iterations. *Crowd counting*: DM-Count's OT loss on 80 crops of 20 annotated crowd images, 100 iterations per image (appendix I).

Baselines. Batched log-domain *POT* 0.9.6 (Flamary et al., 2021), *GeomLoss* 0.3.1 (Feydy et al., 2019), and *OTT-JAX* 0.6.0 (Cuturi et al., 2022) as `jit(vmap(Sinkhorn))` with highest matmul precision (appendix D), plus each application's own implementation: the PyTorch reference loop of mHC (Xie et al., 2025), also compiled with `torch.compile` (Ansel et al., 2024), triton-sinkhorn, a public fused Triton kernel for the mHC layer (LottoLottoLotto, 2026), TileLang-mHC, the Sinkhorn kernel of TileLang's mHC example as benchmarked by mHC-proj (Wang et al., 2025; Wang & Qiu, 2026), and the official modules of OTA (Ge et al., 2021a) and GeoTransformer (Qin et al., 2022).

Accuracy gates. Methods that stop at a tolerance request $\tau = 10^{-4}$. A common FP32 routine, the consumer, rebuilds every plan from the returned potentials, and a run counts only if every instance of every timed call has a two-sided residual of at most $2 \cdot 10^{-3}$, a threshold that admits instances stopped at the cap. Fixed-iteration methods must reproduce the FP64 K-step map: the assignment of every anchor (OTA), forward outputs and input gradients within 10^{-4} and 10^{-3} relative (mHC), or forward outputs within 10^{-4} (patches).

Timing. Each timed call runs from device-resident inputs to the returned output (for table 2, through the common consumer) and includes input checks and preparation; replays of CUDA graphs skip input checks. Each method runs in its own process, with the order rotated across outer blocks, after warm-up until no compilation remains. We report medians over blocks and ratios paired per block (appendix E), on one NVIDIA A100-SXM4-80GB with PyTorch 2.5.1 and CUDA 12.4, in FP32 without TF32.

Table 2: **Time per call** (ms, median of 10 blocks, with validation, preparation, and consumer); OTT/DRAINSINKHORN paired per block, 95% bootstrap CI; all outputs pass the gate of section 5.1. GeomL.: GeomLoss, which takes no tolerance (appendix D); OTT: OTT-JAX. $\leq \tau$: share within $\tau = 10^{-4}$ in FP64, rounded down, DRAINSINKHORN/OTT-JAX. R_{work}: (3).

Workload	R_{work}	DRAINSINKHORN	POT	GeomL.	OTT	OTT/DRAINSINKHORN [CI]	$\leq \tau$ (%)
Converged within the default cap							
`dd-t8`	1.19	4.68	40.7	1220	23.9	5.12 [5.09, 5.15]	99/97
`mot-b64`	1.40	2.68	25.7	1224	9.81	3.68 [3.62, 3.73]	97/95
`dd-t16`	1.19	20.6	109	580	56.3	2.73 [2.72, 2.74]	100/99
`mot-win`	2.59	101	582	10031	306	3.04 [3.02, 3.05]	98/98
Budget-limited at the default cap							
`dd-t4`	2.43	9.58	327	9730	101	10.53 [10.49, 10.62]	84/84
`mot-b16`	3.70	5.85	85.1	3714	62.9	10.75 [10.67, 10.92]	85/88
`mot-b32`	4.70	4.65	83.0	3714	42.6	9.17 [9.01, 9.28]	98/97

5.2 TRAINING LAYERS

Label assignment. OTA's official configuration trains with 16 images per step on 8 GPUs (Ge et al., 2021a), so each GPU assigns 2 images per call. On this call DRAINSINKHORN is the fastest method at 1.88 ms, 13.0× faster than OTA's per-image loop (table 1a). OTT-JAX nearly matches it (1.05×). In a one-GPU training step timed in parts (appendix H), assignment falls from 16.3% to 4.7% of the step and its Sinkhorn part from 12.8% to 1.1%, a 1.13× faster step with no assignment changed.

mHC projections. With the layer replayed from CUDA graphs (table 1b), DRAINSINKHORN is 2.60–2.95× faster than the PyTorch loop under `torch.compile` and 2.54–7.91× faster than triton-sinkhorn on single sublayers, and 1.83–2.60× and 14.3–41.1× with all 16 sublayers per call. TileLang-mHC is 1.47–1.48× faster in the forward pass on single sublayers and 2.08–4.71× with the backward, but its backward, an implicit gradient at the returned matrix, differs from the gradient of the 20-step map by 8–46%. Against the official loop, a whole GPT training step is 1.73× faster with DRAINSINKHORN, 1.32× with `torch.compile`, and 1.80× with TileLang-mHC (appendix I).

Crowd counting. DM-Count (Wang et al., 2020) trains a density map with an OT loss, 100 iterations per image between the 64×64 predicted density and the annotated heads, one image after another. On 80 crops of dense crowds, DRAINSINKHORN makes this loss 3.35× and the training step 1.51× faster (loss share 50% to 24%). The official FP32 solver drops the mass of cells far from every head, so the gradients differ (appendix I).

Patch matching. GeoTransformer pads every patch to 64 points; DRAINSINKHORN solves each instance at its valid size. Inside the model (table 1c), OT becomes 9.94× faster at inference and 10.4× faster in a training step, a 1.18× faster step, and the same 30 of 32 pairs register with either module. On forward calls, where POT and OTT-JAX get the padded 65×65 problems, DRAINSINKHORN is also fastest, 2.80–11.9× faster than OTT-JAX and POT (appendix H). On four pairs, peak memory above the inputs in forward and backward falls from 906 MB (official) to 53 MB (appendix G).

5.3 INFERENCE WORKLOADS

OTT-JAX is the fastest baseline on all seven workloads (table 2), and DRAINSINKHORN is 2.73–10.75× faster than OTT-JAX and 5.3–34× faster than POT. POT stops the whole batch on one error; its median residuals are 6–89× smaller than those of DRAINSINKHORN and OTT-JAX (appendix J). On three workloads some instances reach the default cap under every method. With a fivefold cap at $\tau = 10^{-4}$, DRAINSINKHORN is 29.5× and 37.9× faster than OTT-JAX on `dd-t4` and `mot-b16`, with median residuals within 1% and median depths within one check interval of OTT-JAX's (appendix G); the POT ratio also grows with the cap (figure 4a). On grids, the separable option beats LogSinkhornGPU's cell solver by 11.1–17.6× where both converge (appendix F), and on replayed flow-matching OT batches DRAINSINKHORN is 1.62–1.74× faster than OTT-JAX (appendix K).

5.4 What release buys

We hold each kernel fixed and remove only release (figure 3). On the four resident batches spanning two or three A100 waves, release cuts wall time by 1.10–1.60× and solver time by 1.15–2.01× against the same kernel run to $D_{\max}$ with the same checks; the solver-time ratios lie within the interval of Proposition 1 except on `mot-b32` (2.01 vs. 2.00). `mot-b64` fits in one wave yet gains 1.03×, which the proposition does not explain. On the indexed path (`dd-t16`, `mot-win`), masking finished instances costs 1.33× and 1.88×, and running all to $D_{\max}$ 1.15× and 2.27×.

Where the gains come from. Label assignment gains 2.06× from batching and 6.31× more from the short-by-long kernel; mHC gains from packing, patches from valid-size instances, and none from release. On small instances, resident solves are 13–65× faster than the indexed path.

5.5 Operating range

On synthetic point clouds ($d \le 4$, $N = M \le 512$, $B \in \{1, 16, 1024\}$; figure 4c shows $d = 4$), DrainSinkhorn is faster than OTT-JAX for every $N \le 128$ and for every N at $B = 1024$. With $B \le 16$, OTT-JAX is faster at $N = 512$ and in half of the $N = 256$ cells (OTT-JAX/DrainSinkhorn down to 0.42), and POT only at $N = 512$ with $B = 1$ and in one $B = 16$ cell; all pass the gate.

6 Related work

Fast Sinkhorn solvers. Entropic OT and its log-domain stabilization are standard (Cuturi, 2013; Peyré & Cuturi, 2019; Schmitzer, 2019); online reductions evaluate the cost on the fly (Feydy et al., 2019; Charlier et al., 2021), and FlashSinkhorn fuses the streamed log-sum-exp in an IO-aware kernel (Ye et al., 2026); we build on both. FlashSinkhorn's batched interface loops over problems in Python, so we compare with it on single large problems (appendix F). POT and OTT-JAX batch and vectorize solvers (Flamary et al., 2021; Cuturi et al., 2022), and cuRegOT targets single large problems (Qiu, 2026). Medina & Schmitzer (2025) solve the small cell problems of domain decomposition with a batched GPU solver that uses separable reductions (OT Group Göttingen, 2025) and stops a batch on an aggregate error; we adopt the reduction and stop per instance. Training layers motivate shape-specific kernels: mHC fuses its $n = 4$ projection into one kernel (Xie et al., 2025); Wang & Qiu (2026) solve each 4×4 problem in half a warp's registers, and a finished problem waits for its block.

Batched execution of problems with different lengths. Finished items leave a batch in iteration-level scheduling of generation (Yu et al., 2022), automatic batching (Radul et al., 2020), batched ODE solvers (Lienen & Günnemann, 2022), interior-point solvers (Viljoen et al., 2026), and batched linear solvers (Nguyen et al., 2023). To our knowledge, no prior batched OT solver removes converged instances from later launches so that the grid narrows. Implicit gradients (Eisenberger et al., 2022; Pineda et al., 2022) target the optimum, not the finite map.

7 Limitations and conclusion

Making the instance the unit of GPU execution serves inference and training alike: DrainSinkhorn is the fastest compared method on every tracking and domain-decomposition workload, up to 10.75× and 34× faster than OTT-JAX and POT, and cuts the Sinkhorn part of an OTA step, which YOLOX dropped, from 12.8% to 1.1%.

Label-assignment inputs come from a converged detector, and timings cover single training steps; mHC inputs come from a small model, and patch inputs from 32 3DMatch pairs. 4×4 projection kernels remain faster, mostly in the backward pass; full 65×65 patches need 128×128 tiles; packing covers fixed-iteration solves only. DrainSinkhorn loses to OTT-JAX on up to 16 clouds of 512 points and in half of the 256-point cells (section 5.5), and to OTT-JAX or POT on 7 of 9 single large problems, by 10.9–16.7× at $d = 3072$, where it recomputes distances per half-step (appendix F).

REPRODUCIBILITY STATEMENT

Appendices C and D describe the workloads and the configuration of every baseline, appendix E lists the timing protocol and the solver commit and Triton version behind each table, and appendix H describes how the training-layer inputs were captured. The supplementary material contains the anonymized source code of DRAINSINKHORN at the final solver commit, with its tests.

USE OF LARGE LANGUAGE MODELS

The research idea, the problem setting, and the overall design of the solver and of this paper were provided by the author. Large language models (LLMs) were used in four roles. First, LLM coding agents wrote the GPU kernels of DRAINSINKHORN and the benchmark code, including the baseline wrappers and the timing and comparison code; the author reviewed all of this code. Second, LLM coding agents wrote the remaining experiment code, including run scheduling and analysis scripts; LLMs reviewed it, and the author read the code behind the key steps of diagrams in which an LLM mapped each code block. Third, the author drafted the outline of the paper, an LLM wrote the text from that outline and from the experimental records, and the author revised the result into the final version. Fourth, LLM-based search tools were used to find related work: they listed candidate papers and suggested that some could be built into the solver; the author discarded some, tested the rest, and adopted the separable reduction of section 3.2 (Medina & Schmitzer, 2025; OT Group Göttingen, 2025). The author takes full responsibility for the content of the paper.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

Mauro Bonafini and Bernhard Schmitzer. Domain decomposition for entropy regularized optimal transport. *Numerische Mathematik*, 149:819–870, 2021.

Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the GPU, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74):1–6, 2021.

Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transportation distances. In *Advances in Neural Information Processing Systems*, volume 26, pp. 2292–2300, 2013. URL https://arxiv.org/abs/1306.0895.

Marco Cuturi, Laetitia Meng-Papaxanthos, Yingtao Tian, Charlotte Bunne, Geoff Davis, and Olivier Teboul. Optimal transport tools (OTT): A JAX toolbox for all things wasserstein, 2022. URL https://arxiv.org/abs/2201.12324.

Marvin Eisenberger, Aysim Toker, Laura Leal-Taixé, Florian Bernard, and Daniel Cremers. A unified framework for implicit Sinkhorn differentiation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 509–518, 2022.

Jean Feydy, Thibault Séjourné, François-Xavier Vialard, Shun ichi Amari, Alain Trouvé, and Gabriel Peyré. Interpolating between optimal transport and MMD using sinkhorn divergences. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pp. 2681–2690, 2019. URL https://proceedings.mlr.press/v89/feydy19a.html.

Rémi Flamary, Nicolas Courty, Alexandre Gramfort, Mokhtar Z. Alaya, Aurélie Boisbunon, Stanislas Chambon, Laetitia Chapel, Adrien Corenflos, Kilian Fatras, Nemo Fournier, Léo Gautheron, Nathalie T. H. Gayraud, Hicham Janati, Alain Rakotomamonjy, Ievgen Redko, Antoine Rolet, Antony Schutz, Vivien Seguy, Danica J. Sutherland, Romain Tavenard, Alexander Tong, and Titouan Vayer. POT: Python optimal transport. *Journal of Machine Learning Research*, 22(78): 1–8, 2021. URL https://www.jmlr.org/papers/v22/20-451.html.

Zheng Ge, Songtao Liu, Zeming Li, Osamu Yoshie, and Jian Sun. OTA: Optimal transport assignment for object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021a.

Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. YOLOX: Exceeding YOLO series in 2021. *arXiv preprint arXiv:2107.08430*, 2021b.

Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

Marten Lienen and Stephan Günnemann. torchode: A parallel ODE solver for PyTorch, 2022. URL https://arxiv.org/abs/2210.12375.

Philipp Lindenberger, Paul-Edouard Sarlin, and Marc Pollefeys. LightGlue: Local feature matching at light speed. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

LottoLottoLotto. triton-sinkhorn. https://github.com/LottoLottoLotto/triton-sinkhorn, 2026. Commit c5bd2f6d, accessed 2026-09-25.

Ismael Medina and Bernhard Schmitzer. Flow updates for domain decomposition of entropic optimal transport. *ESAIM: Mathematical Modelling and Numerical Analysis*, 59(3), 2025. doi: 10.1051/m2an/2025022.

Anton Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. MOT16: A benchmark for multi-object tracking, 2016. URL https://arxiv.org/abs/1603.00831.

Phuong Nguyen, Pratik Nayak, and Hartwig Anzt. Porting batched iterative solvers onto Intel GPUs with SYCL, 2023. URL https://arxiv.org/abs/2308.08417.

OT Group Göttingen. LogSinkhornGPU: GPU log-domain Sinkhorn solvers for domain decomposition. https://github.com/OTGroupGoe/LogSinkhornGPU, 2025. Commit 58ac43cc.

Gabriel Peyré and Marco Cuturi. Computational optimal transport. *Foundations and Trends in Machine Learning*, 11(5–6):355–607, 2019.

Luis Pineda, Taosha Fan, Maurizio Monge, Shobha Venkataraman, Paloma Sodhi, Ricky T. Q. Chen, Joseph Ortiz, Daniel DeTone, Austin Wang, Stuart Anderson, Jing Dong, Brandon Amos, and Mustafa Mukadam. Theseus: A library for differentiable nonlinear optimization. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Zheng Qin, Hao Yu, Changjian Wang, Yulan Guo, Yuxing Peng, and Kai Xu. Geometric transformer for fast and robust point cloud registration. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

Yixuan Qiu. cuRegOT: A GPU-accelerated solver for entropic-regularized optimal transport, 2026. URL https://arxiv.org/abs/2605.08793.

Alexey Radul, Brian Patton, Dougal Maclaurin, Matthew D. Hoffman, and Rif A. Saurous. Automatically batching control-intensive programs for modern accelerators. In *Proceedings of Machine Learning and Systems*, volume 2, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/hash/39945d578f616735572174bf5e8f155d-Abstract.html.

Bernhard Schmitzer. Stabilized sparse scaling algorithms for entropy regularized transport problems. *SIAM Journal on Scientific Computing*, 41(3):A1443–A1481, 2019.

Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. FCOS: Fully convolutional one-stage object detection. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Unidata. Crowd density dataset – different crowd sizes (preview). Hugging Face dataset `UniDataPro/crowd-counting`, CC BY-NC-ND 4.0, 2024.

John Viljoen, Johanna Haffner, Masayoshi Tomizuka, and Negar Mehr. Scaling nonlinear optimization: Many problems one GPU, 2026. URL `https://arxiv.org/abs/2606.26341`.

Boyu Wang, Huidong Liu, Dimitris Samaras, and Minh Hoai. Distribution matching for crowd counting. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Chenrui Wang and Yixuan Qiu. Accelerating Birkhoff projection for manifold-constrained hyper-connections, 2026. URL `https://arxiv.org/abs/2606.07574`.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. TileLang: A composable tiled programming model for AI systems. *arXiv preprint arXiv:2504.17577*, 2025.

Zhenda Xie, Yixuan Wei, Huanqi Cao, Chenggang Zhao, Chengqi Deng, Jiashi Li, Damai Dai, Huazuo Gao, Jiang Chang, Kuai Yu, Liang Zhao, Shangyan Zhou, Zhean Xu, Zhengyan Zhang, Wangding Zeng, Shengding Hu, Yuqing Wang, Jingyang Yuan, Lean Wang, and Wenfeng Liang. mHC: Manifold-constrained hyper-connections. *arXiv preprint arXiv:2512.24880*, 2025.

Felix X.-F. Ye, Xingjie Li, An Yu, Ming-Ching Chang, Linsong Chu, and Davis Wertheimer. FlashSinkhorn: IO-aware entropic optimal transport on GPU, 2026. URL `https://arxiv.org/abs/2602.03067`.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538, 2022. URL `https://www.usenix.org/conference/osdi22/presentation/yu`.

Hao Yu, Fu Li, Mahdi Saleh, Benjamin Busam, and Slobodan Ilic. CoFiNet: Reliable coarse-to-fine correspondences for robust point cloud registration. In *Advances in Neural Information Processing Systems*, 2021.

Jiahao Zhang, Rui Xue, Wenqi Fan, Xin Xu, Qing Li, Jian Pei, and Xiaorui Liu. Linear-time graph neural networks for scalable recommendations. In *Proceedings of the ACM Web Conference (WWW)*, 2024.

A PROOFS

Throughout, arithmetic is exact.

Proof of (6). $\sum_t |\mathcal{A}_t| = \sum_t \sum_b \mathbf{1}[D_b \geq t] = \sum_b \sum_{t=1}^{D_{\max}} \mathbf{1}[t \leq D_b] = \sum_b D_b.$ □

Proof of Proposition 1. Write $p_b = \theta D_b + \sigma$ and $W = \sum_b p_b$. At most S programs run at once, so $T_{\text{active}} \geq W/S$, and no program finishes before $p_{\max} = \theta D_{\max} + \sigma$ has elapsed. For the upper bound, let program j finish last and start at s_j. Because the scheduler is work-conserving and j was pending until s_j, all S slots were busy throughout $[0, s_j)$, so $Ss_j \leq W - p_j$. Then $T_{\text{active}} = s_j + p_j \leq W/S + (1 - 1/S)p_j \leq W/S + (1 - 1/S)p_{\max}$ (Graham, 1969). In the static control all B programs have length $p_{\max}$; a work-conserving schedule of equal jobs runs them in $\lceil B/S \rceil$ waves, so $T_{\text{static}} = \lceil B/S \rceil p_{\max}$. If $B \leq S$, $T_{\text{static}} = p_{\max} \leq T_{\text{active}}$. If the depths are drawn from a fixed distribution with mean $\bar{D}$ and $B \to \infty$, both T_{static} and T_{active} grow like B/S times $p_{\max}$ and $\theta\bar{D} + \sigma$ respectively. □

The assumptions exclude contention between concurrent programs. Each resident program loads its tile once and then works on chip, so memory traffic is limited to the initial loads and final stores, but co-resident programs on one SM still share its issue bandwidth.

B IMPLEMENTATION DETAILS

Resident path *one program per instance b, B programs in one launch*
load C_b, a_b, b_b on chip; $f, g \leftarrow 0$; $t \leftarrow 0$
repeat
  apply (2) $\min(q, K_{\max} - t)$ times; $t \leftarrow t + q$
  $P \leftarrow \exp((f \oplus g - C_b)/\varepsilon_b)$ on chip; $r \leftarrow \max(\|P\mathbf{1} - a_b\|_1, \|P^\top \mathbf{1} - b_b\|_1)$
until $r \le \tau$ **or** $t \ge K_{\max}$
write $f_b, g_b, r_b, D_b \leftarrow t$ and exit *the scheduler reuses the slot*
Indexed path *host loop over launches*
$\mathcal{A} \leftarrow \{1, \dots, B\}$; $f, g \leftarrow 0$; *reuse*←false
for $t = 1, \dots, K_{\max}$ **while** $\mathcal{A} \neq \emptyset$ **do**
  $f_{\mathcal{A}} \leftarrow f^{\mathrm{bal}}_{\mathcal{A}}$ **if** *reuse* **else** row update *survivors reuse the screen's update*
  $g_{\mathcal{A}} \leftarrow$ column update; *reuse*←false
  if $t \equiv 0 \pmod q$ **then**
    $f^{\mathrm{bal}}_{\mathcal{A}}, s_{\mathcal{A}} \leftarrow$ row update from current g, with screen epilogue (5)
    $V \leftarrow \{b \in \mathcal{A} : s_b \le \tau\}$; $r_V \leftarrow$ both residuals of current (f_V, g_V)
    store (f_b, g_b, r_b, t) for accepted $b \in V$; $\mathcal{A} \leftarrow \mathcal{A} \setminus \{b \in V : r_b \le \tau\}$; *reuse*←true
  end if
end for
verify remaining $\mathcal{A}$ once and mark them capped

Algorithm 1: Per-instance release (both paths return f, g, r, D, status in original order)

Interface. The public entry point accepts dense costs, point clouds, and lists of problems; the inference workloads use point clouds and the training layers dense costs. All execution paths return one result type with per-instance potentials, residuals, depths, and status, so a consumer does not need to know which path ran. All kernels are written in Triton and run in strict FP32 with TF32 disabled.

Routing. A tolerance-mode FP32 CUDA solve without gradients uses the resident path if both supports have between 1 and 64 points; squared-Euclidean supports with $d \le 4$ and up to 512 points use the low-dimensional resident path when its tiling search (section 3.1) selects it. A fixed-iteration dense FP32 CUDA solve with a per-instance cost uses the packed resident map if both sides have at most 128 points, or if the short side has at most 64 and the padded tile at most 2^{17} elements; it supports gradients with respect to all inputs. Other calls use the indexed path. The separable option is selected explicitly by passing the grid shapes of both supports, each axis with at most 16 points. On the A100, the compiled resident kernel admits at most 1296, 972, and 324 concurrent programs for 16-, 32-, and 64-point instances; these are the values of S used in figure 3a.

Cost preparation. Coordinates are centered per instance and squared norms are cached once per call. The squared distance is computed as $\|x\|^2 + \|y\|^2 - 2\langle x, y\rangle$ in FP32 without TF32. For the resident path the tile is materialized once; a support shared by all instances is stored once and read with batch stride zero. The separable option stores only the two axis cost tiles of each instance.

Indexed kernels. Each half-step launches a grid over (active instance, row tile); a program streams column tiles, computes the distance tile by a tiled dot product, and maintains an online log-sum-exp. The dot-product and online log-sum-exp helpers are adapted from FlashSinkhorn (Ye et al., 2026) under its MIT license. Tile shapes are chosen by autotuning keyed on instance size and active width. The screen is a flag of the row-update kernel that additionally accumulates $a_i |\operatorname{expm1}((f_i - f^{\mathrm{bal}}_i)/\varepsilon)|$; a separate kernel computes both marginals for verification.

Fixed-iteration kernels. The packed forward kernel of section 3.3 holds 32 instances per program for padded tiles of at most 64 elements and $\max(1, \lfloor 2048/(RC) \rfloor)$ for larger tiles of $R \times C$ elements, with four warps. Input value checks (finite cost, nonnegative marginals of equal mass, positive ε) run as one packed kernel whose error flags are read once; `validate=False` skips them. Tests compare the forward outputs and the vector–Jacobian products with respect to C, a, b, and ε of the packed path with FP64 automatic differentiation of the same K-step map. The row and column residuals of the returned iterate are computed by one more packed kernel.

C WORKLOADS

Table 3 lists the seven inference workloads.

Table 3: **Workloads.** All instances are balanced squared-Euclidean EOT problems with $\tau = 10^{-4}$ and a check every $q = 5$ iterations. B is the number of instances per batch. MOT17 buckets pad each detection set to the bucket size with points of mass 10^{-8}; domain-decomposition tiles share one grid support.

Workload	Source	B	$N{=}M$	d	ε	$K_{\max}$	Supports	Path
`mot-b16`	MOT17 frame pairs	3771	16	4	0.05	500	independent	resident
`mot-b32`	MOT17 frame pairs	1469	32	4	0.05	500	independent	resident
`mot-b64`	MOT17 frame pairs	69	64	4	0.05	500	independent	resident
`mot-win`	MOT17 windows	64	1024	4	0.05	500	independent	indexed
`dd-t4`	DD tile pool	2304	16	2	0.01	2000	shared grid	resident
`dd-t8`	DD tile pool	576	64	2	0.01	500	shared grid	resident
`dd-t16`	DD tile pool	144	256	2	0.01	500	shared grid	resident

D BASELINE CONFIGURATION

POT. `ot.batch.solve_batch` with `solver="log_sinkhorn"`, cost from squared distances, the workload cap as `max_iter`, and `tol`$= \tau/\sqrt{N}$. POT stops on the ℓ_2 norm of the row-marginal error of the whole batch; since $\|r\|_1 \le \sqrt{N}\|r\|_2$, this tolerance cannot stop before the ℓ_1 gate would accept.

GeomLoss. `geomloss.ot.solve_batch` on the dense squared-distance cost with `reg`$= \varepsilon$. Version 0.3.1 rejects a tolerance; its iteration count drives an ε-annealing schedule. We therefore hold it only to the common gate and give each workload the smallest count on the grid 500, 1000, 2000, 3000, 5000, 7500, 10,000, 15,000, 20,000, 30,000, 40,000 that passes it: 40,000 for `dd-t4`, 15,000 for `mot-b16` and `mot-b32`, 5000 for `dd-t8` and `mot-b64`, 3000 for `mot-win`, and 1000 for `dd-t16`.

OTT-JAX. `jax.jit(jax.vmap(solve))`, where `solve` builds a `PointCloud` geometry for one instance and runs `Sinkhorn(lse_mode=True, threshold=`τ`, norm_error=1, inner_iterations=`q`, max_iterations=`$K_{\max}$`)`. We set `jax_default_matmul_precision` to `highest`, because the default can use TF32. In the fixed-iteration cells (table 1a,c), `min_iterations` equals `max_iterations` and the timed call returns only the plan: asking for the iteration count as well keeps the convergence checks, which XLA otherwise removes, and made each call up to 1.7× slower. The count is read in a separate untimed call.

LogSinkhornGPU. `LogSinkhornCudaImageOffset` (commit 58ac43cc), with `max_error=` τ, `inner_iter=` q, the workload cap as `max_iter`, and `iterate_until_max_error`. It requires equispaced grid axes, which the DD tiles satisfy. Its potentials are converted to our convention by absorbing the reference measures, $f = \alpha + \varepsilon \log a$ and $g = \beta + \varepsilon \log b$. We time the cell solver on the tile pools, not the outer domain-decomposition iterations.

Common consumer. For every method, the plan is reconstructed from the returned potentials in FP32 blocks, and the gate and the transport cost are computed from it identically.

E MEASUREMENT PROTOCOL

Each outer block runs one process per method on the same GPU, with the method order rotated across blocks. Inside a process, the method is called until no compilation or autotuning occurs, then timed

Table 4: **Regular-grid tiles:** DRAINSINKHORN with the separable option against the native CUDA solver of LogSinkhornGPU (OT Group Göttingen, 2025) (ms, 10 paired blocks, native/DRAINSINKHORN with 95% CI). *Capped*: instances at the cap, DRAINSINKHORN/native.

Tiles	DRAINSINKHORN	Native	Ratio [95% CI]	Capped
`dd-t4`	9.70	625	64.5 [63.8, 65.2]	343/2304
`dd-t8`	3.54	62.5	17.6 [17.3, 17.8]	0/0
`dd-t16`	5.29	58.7	11.1 [11.0, 11.2]	0/0

20 times; each call starts from zero potentials and shares no state with previous calls. Wall-clock time is taken after device synchronization, and device events record the solver and consumer intervals. For each workload and method, the block statistic is the median over repetitions; tables report the median over blocks, and intervals come from bootstrap resamples of blocks with paired ratios. Each timed job has a GPU to itself; other jobs may run on the other GPUs of the host (two Xeon 8362, 128 cores, load average at most about 13). The label-assignment runs had the host to themselves; an earlier run that shared the host gave ratios within 1% of them. Records of every run include the GPU UUID, software versions, and commands.

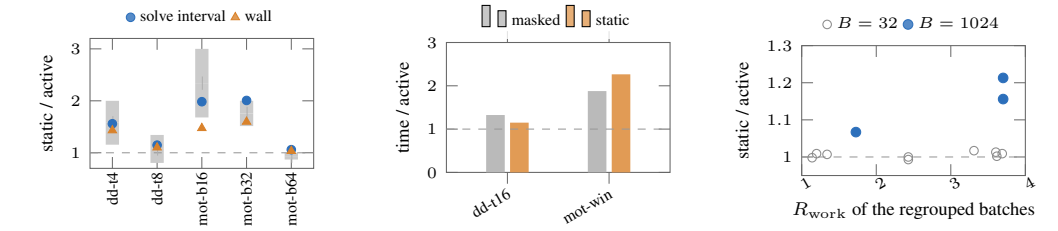
Solver commits. Each table comes from one solver commit and one Triton version. Tables 1, 6 and 7 use the final commit `7a291287`. Table 2, the point-cloud sweep of figure 4c, and the FP64 audit of table 2 use `09579cfa`; the later commits change only the packed fixed-iteration kernel for tiles of at most 64 entries, the short-by-long kernel, and the host-side preparation of a scalar ε, which yields the same values; these workloads use none of the kernels. Table 4, figure 3, the cap sweep, the profile, and the single-problem runs use the earlier commit `42566416`, which ran `dd-t16` on the indexed path, with Triton 3.1.0; all other runs use Triton 3.7.1. Run under the same Triton 3.7.1 on the A100, `42566416` and `09579cfa` return bitwise identical potentials, residuals, iteration counts, plans, and transport costs on the five workloads of table 2 that are resident under both commits; the separable kernel has the same arithmetic in both and differs only in launch configuration. Resident outputs are also bitwise independent of batch composition: permuting a batch or solving a subset returns the same bits for every instance (tested for the packed fixed-iteration and the tolerance resident paths). The short-by-long path picks its chunk size, and the low-dimensional resident path its tiling, from the whole call, so on those paths the independence holds in exact arithmetic only.

Training layers. Each cell times one call of each method, every method in its own process, in 5 outer blocks of 10 calls after 3 warm-up calls, with the method order rotated. mHC methods share logits and an upstream gradient, and forward and forward+backward are timed separately. Patch timings inside the model follow appendix H. The mHC gate compares forward outputs and input gradients with an FP64 evaluation of the same 20-step map; on every mHC cell, DRAINSINKHORN's outputs differ from it by at most $3 \cdot 10^{-6}$ per entry and its input gradients by at most $3 \cdot 10^{-6}$ in relative ℓ_2 norm.

F ADDITIONAL INFERENCE RESULTS

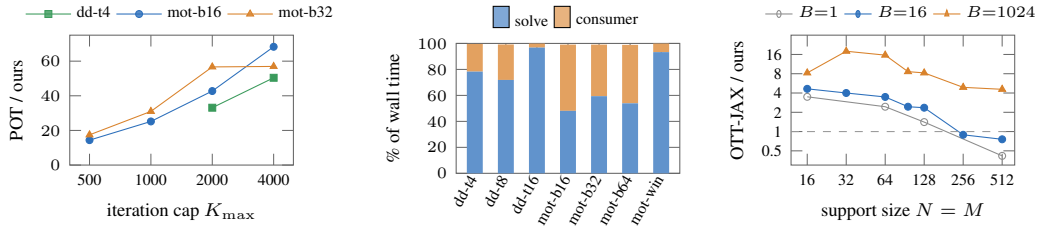
Launches. Under Nsight Systems, one `mot-b16` call issues 70 kernel launches for DRAINSINKHORN against 2504 for OTT-JAX and 11530 for POT, with 2.1 ms of kernel time against 39.1 and 53.8 ms.

Single large problems. For one problem ($B = 1$) with $N \in \{1024, 4096, 16384\}$ and $d \in \{3, 64, 3072\}$, DRAINSINKHORN is slower than the fastest of POT, OTT-JAX, and FlashSinkhorn in 7 of 9 cases, by 10.9–16.7× at $d = 3072$, and faster only for $d = 3$ at $N = 4096$ and 16384 (1.28 and 1.57×). FlashSinkhorn runs with its own early stop, a threshold of $\tau\varepsilon$ on the change of the potentials checked every 5 iterations; it is never the fastest, and its stop returns residuals of $7 \cdot 10^{-7}$ to $7 \cdot 10^{-6}$. These runs use one fresh process per method and 20 repeats (5 at $N = 16384$), without paired intervals.



(a) Release, resident kernel fixed. (b) Release on the indexed path. (c) Depth heterogeneity and waves.

Figure 3: **What release buys.** Solver commit `42566416`, which ran `dd-t16` on the indexed path (appendix E). All ratios are paired over 10 blocks; above the dashed line, release is faster. (a) Resident kernel with release against the same kernel running all instances to $D_{\max}$ with the same checks; gray bars: range predicted by Proposition 1 with $\sigma = 0$. (b) Indexed path with identical iterates: finished instances masked, or all run to $D_{\max}$. (c) Fixed pools of instances regrouped into batches by depth (homogeneous, random, heterogeneous): `dd-t4`, `mot-b16`, and `mot-b32` pools with $B = 32$, which fits in one wave, and a `mot-b16` pool with $B = 1024$.



(a) Iteration budget. (b) Solver and consumer intervals. (c) Support size.

Figure 4: **Budgets, overheads, and scope.** (a), (b) use the earlier solver commit of appendix E, which ran `dd-t16` on the indexed path. (a) Paired POT/DRAINSINKHORN time as the cap grows; `dd-t4` fails the common gate at caps 500 and 1000 for both methods and has no eligible ratio there. (b) Device intervals of the solver call and the common consumer, as shares of the wall time of that run. (c) Paired OTT-JAX/DRAINSINKHORN time on synthetic $d = 4$ point clouds of equal size, tolerance mode; below 1, OTT-JAX is faster. $B = 1$ was run at $N \in \{16, 64, 128, 512\}$.

Regular grids. The separable option is 11.1–17.6× faster than the native LogSinkhornGPU solver on `dd-t8` and `dd-t16`, where both converge within the cap (table 4). Both use the same reduction, so the difference lies in execution and stopping: one program per tile with per-instance release, against batch-wide launches until an aggregate error passes. On `dd-t4` the aggregate stop runs all 2304 tiles to the cap while DRAINSINKHORN releases all but 343, so the 64.5× ratio includes a different amount of work.

Depth heterogeneity. Regrouping fixed pools of instances by depth changes R_{work} but no instance (figure 3c). With $B = 32$, below the 108 SMs of the GPU, every grouping gives 0.99–1.02×. With $B = 1024$ on `mot-b16`, the ratio rises with R_{work}, from 1.07× ($R_{\text{work}} = 1.73$) to 1.21× ($R_{\text{work}} = 3.70$), although the batch fits in one wave ($S = 1296$), where Proposition 1 predicts no gain. The likely gap is its assumption that programs do not slow each other down: up to 12 programs share an SM, and an exiting program leaves the SM to the others.

G SUPPLEMENTARY MEASUREMENTS AT THE FINAL COMMIT

All runs in this section use solver commit `7a291287` with Triton 3.7.1 on one A100 and the protocol of appendix E.

Tolerance. Table 5 repeats two of the three budget-limited workloads of table 2 at four tolerances with the cap raised fivefold, so that fewer than 3% of instances reach it under DRAINSINKHORN or

OTT-JAX. DRAINSINKHORN is 16–38× faster than OTT-JAX and 32–94× faster than POT; POT, which stops the whole batch, runs to the cap at $\tau \leq 10^{-4}$ on `dd-t4` and $\tau \leq 3 \cdot 10^{-4}$ on `mot-b16`. The median residuals of DRAINSINKHORN and OTT-JAX agree within 1% and their median depths within one check interval in every cell. Appendix J reports the share of instances within τ in FP64 at $\tau = 10^{-4}$.

Table 5: Tolerance sweep. Time per call of DRAINSINKHORN (ms) and paired ratios per block (median over 5 blocks, range in brackets). D: median depth of DRAINSINKHORN; capped: instances at the cap, DRAINSINKHORN / OTT-JAX.

Workload	$K_{\max}$	τ	DRAINSINKHORN	D	capped	OTT/DRAINSINKHORN	POT/DRAINSINKHORN
`dd-t4`	10000	10^{-3}	9.8	465	0/0	18.8 [18.5, 19.2]	55.4 [53.9, 56.1]
`dd-t4`	10000	$3 \cdot 10^{-4}$	15.8	520	0/0	32.6 [32.5, 32.9]	93.8 [90.1, 102.0]
`dd-t4`	10000	10^{-4}	19.9	565	16/16	29.5 [28.9, 32.7]	81.4 [79.5, 91.0]
`dd-t4`	10000	$3 \cdot 10^{-5}$	20.9	620	53/54	28.2 [27.1, 31.5]	77.0 [75.6, 88.4]
`mot-b16`	2500	10^{-3}	5.2	30	0/0	16.3 [16.1, 16.5]	32.4 [31.9, 35.4]
`mot-b16`	2500	$3 \cdot 10^{-4}$	6.8	60	0/0	33.7 [33.5, 33.8]	59.6 [58.9, 60.9]
`mot-b16`	2500	10^{-4}	8.0	85	3/5	37.9 [37.5, 38.0]	53.9 [50.4, 55.3]
`mot-b16`	2500	$3 \cdot 10^{-5}$	10.0	135	80/69	30.3 [30.3, 30.9]	41.1 [40.7, 44.5]

Check interval. At the default caps and $\tau = 10^{-4}$, checking every $q \in \{1, 2, 10, 20\}$ iterations takes 12.8, 11.0, 9.6, and 9.4 ms on `dd-t4` and 6.7, 6.2, 5.9, and 5.9 ms on `mot-b16`. Checking every iteration thus costs at most 1.36× over the fastest interval.

Release at the final commit. Rerun at `7a291287`, the release ablation of figure 3 gives the same wall-time ratios within 0.04: running the resident kernel to $D_{\max}$ with the same checks costs 1.11–1.60× on the four multi-wave batches and 1.04× on `mot-b64`; on the indexed path, which this ablation selects explicitly, masking costs 1.31× and 1.85× and running every instance to $D_{\max}$ 1.15× and 2.23× on `dd-t16` and `mot-win`.

Memory of the fixed-iteration layers. Peak memory allocated during one forward and backward call, above the inputs: for the mHC layer ($n = 4$, 20 iterations), DRAINSINKHORN needs 1.07 GB at $B = 2^{20}$ against 3.76 GB for the PyTorch reference loop and 2.84 GB compiled; the reference loop needs 3.4–3.5 times the memory of DRAINSINKHORN at $B = 4096$, 65,536, and 2^{20}. For GeoTransformer's patch matching (256 instances, 100 iterations), the official module needs 906 MB and DRAINSINKHORN 53 MB on each of four pairs.

Label assignment with one padded shape. A separate run adds OTT-JAX with every call padded to the largest shape of the run. It then takes 3.47 ms per call, 1.84× (blocks 1.62–1.91) DRAINSINKHORN's 1.89 ms; with per-call shapes, OTT-JAX takes 1.05× (blocks 0.92–1.07) in the same run. On all 16 images of a step, the ratios are 2.56× and 2.14×. No method changes an assignment.

H TRAINING LAYERS: CAPTURE AND ADDITIONAL RESULTS

Label-assignment capture. We follow OTA's official assignment code (repository commit `2c85b4d`): per image, the cost is the focal classification loss plus 1.5 times the IoU loss of the predicted boxes, plus 10^6 outside the center-sampling region, with a background row; supplies are the dynamic k of each box (the sum of its top-20 IoUs) and the remaining anchors for the background, and every anchor demands one unit. Predictions come from torchvision's COCO-trained FCOS (Tian et al., 2019), which has the head OTA builds on, at OTA's training resolution (short side 800, long side at most 1333), on COCO val2017 images read from a parquet copy of the dataset whose images and boxes we checked to be bit-identical to the official files. Deviations: the checkpoint is converged rather than taken from an OTA training run; FCOS's box decoder and normalization are used as the weights require; crowd boxes are dropped; the P6/P7 point spacing follows torchvision's anchor generator. A step has 16 instances of 2 to 54 rows, padded to 26,863 to 35,810 anchors; one GPU's share, 2 images, is padded as a pair (13,868 to 34,025 anchors). Every captured instance stores

the official solver's plan as a reference. Every FP32 implementation other than DRAINSINKHORN, the official one included, differs from the FP64 map by up to 0.14–0.21 in single plan entries on some calls, and POT changes some assignments on the whole step; DRAINSINKHORN stays within $6.4 \cdot 10^{-6}$ with the cost shift of section 3.4. The assignment arms in table 1a: the official per-image `SinkhornDistance`; the same updates on the step padded into one tensor; DRAINSINKHORN on transposed views, which reproduces OTA's column-first order; POT's batched log-domain solver and OTT-JAX's `jit(vmap(Sinkhorn))` on the padded step with exactly 50 iterations. The step share times torchvision FCOS's forward, loss, and backward for the images of one call (2 or 16; TF32 off) and adds the assignment timed separately.

mHC capture. We implemented the mHC residual from the equations of Xie et al. (2025) ($n = 4$ streams, one connection per attention and MLP sublayer, 20 Sinkhorn iterations) in an 8-block GPT of width 384 (34M parameters) and trained it for 2,500 steps of 16K tokens on FineWeb-edu (held-out loss 10.9 to 4.6). At steps 0, 100, 500, 1,250, and 2,500 we recorded, for 4,096 held-out tokens, every sublayer's Sinkhorn input logits and the upstream gradient of its output. The logit standard deviation grows from 0.013 at initialization to between 0.3 and 4.2 across sublayers; at step 2,500, 44% of all instances, and 98–100% in the hardest sublayers, have a two-sided residual above 10^{-4} after 20 iterations. Table 1b uses the hardest (std 4.2), a medium (1.6), and an easy (0.3) sublayer, and all 16 sublayers in one call. TileLang-mHC's forward fails the gate only on the 16-sublayer call (error $1.2 \cdot 10^{-4}$), on a sublayer not shown separately; on the three single sublayers its error is at most $3 \cdot 10^{-5}$.

Patch-matching capture. We run GeoTransformer (repository commit `e7a135a`) with its released 3DMatch weights on the first four pairs of each of the eight test scenes in the official pair list, 32 pairs of 40 fragments, downsampled with 2.5 cm voxels. One deviation: the KPConv neighbor limits, which the official script estimates from the training split, are estimated the same way from these fragments. A hook records the inputs of the fine-matching Sinkhorn: per pair, 256 score matrices of 64×64, the row and column masks, and the learned dustbin score (6.29). DRAINSINKHORN replaces the module behind the same interface: it moves the dustbin of each instance next to its valid points, so the instance fits a 64×64 tile, gives the remaining entries zero mass, runs the 100 iterations, and writes the log-plan back into the official layout; the instance is the same up to a relabelling, so the 100-step map is the same. Inside the model, CUDA events time the OT module's forward and, in a training step (training-mode forward, the official loss, backward), its backward, from the gradient reaching its output to the gradient reaching its input; each module runs 3 blocks of 5 steps per pair, each after 2 warm-up steps, alternating which runs first, and training-mode sampling is reseeded so both see the same targets. Estimated poses differ by at most $9.5 \cdot 10^{-7}$ and losses by $1.2 \cdot 10^{-7}$ relative. POT and OTT-JAX receive the official padded 65×65 instances with zero mass on masked points (OTT-JAX transposed, since it updates the column potential first), and their time includes building them and converting the plan to the official output. Their backward passes, by automatic differentiation through the iterations, took about 23 ms on two pairs in a probe, longer than their forward passes, so we compare them on forward calls only (5 blocks of 10 calls, one pair per call).

Synthetic mHC inputs. Tables 6 and 7 repeat table 1b with synthetic logits $\sim \mathcal{N}(0, \sigma^2)$ and up to 1M instances per call.

Table 6: table 1b with synthetic logits, $\sigma = 1$. *Computes a different quantity (table 1).

Input	B	DRAINSINKHORN (ms)	`torch.compile`	triton-sinkhorn	TileLang-mHC*
$\mathcal{N}(0, 1^2)$, 4×4	16K	0.073 / 0.274	2.66 / 2.87	21.9 / 7.55	**0.84 / 0.31**
$\mathcal{N}(0, 1^2)$, 4×4	128K	0.265 / 0.763	1.53 / 2.37	46.7 / 16.7	1.07 / **0.54**
$\mathcal{N}(0, 1^2)$, 4×4	1M	1.85 / 5.29	1.85 / 3.27	43.8 / 19.1	1.15 / **0.57**
$\mathcal{N}(0, 1^2)$, 8×8	16K	0.186 / 0.722	1.82 / 1.22	43.6 / 16.1	–
$\mathcal{N}(0, 1^2)$, 8×8	128K	1.03 / 4.17	1.74 / 1.19	51.2 / 22.0	–

mHC with hard inputs. Table 7 repeats table 1b with $\sigma = 10$, the setting of Wang & Qiu (2026). There, at least 97% of the instances are above the tolerance after 20 iterations, and TileLang-mHC

fails the same-map gate, while the 20-step map keeps its column marginal exact (DRAINSINKHORN: $7 \cdot 10^{-6}$).

Table 7: table 1b with logits $\sim \mathcal{N}(0, 10^2)$. †Fails the same-map gate. *Computes a different quantity (table 1).

Input	B	DRAINSINKHORN (ms)	`torch.compile`	triton-sinkhorn	TileLang-mHC*
$\mathcal{N}(0, 10^2)$, 4×4	16K	0.074 / 0.274	2.64 / 2.79	21.8 / 7.55	**0.83**† / **0.30**†
$\mathcal{N}(0, 10^2)$, 4×4	128K	0.266 / 0.761	1.52 / 2.38	46.7 / 16.7	1.07† / **0.54**†
$\mathcal{N}(0, 10^2)$, 4×4	1M	1.85 / 5.29	1.85 / 3.27	43.8 / 19.1	1.15† / **0.57**†
$\mathcal{N}(0, 10^2)$, 8×8	16K	0.186 / 0.720	1.82 / 1.22	43.6 / 16.1	–
$\mathcal{N}(0, 10^2)$, 8×8	128K	1.03 / 4.17	1.74 / 1.19	51.2 / 22.0	–

mHC-proj. The 4×4 Sinkhorn kernel of mHC-proj (Wang & Qiu, 2026) (commit `b85c9ab`) launches every kernel on `cudaStreamPerThread` instead of PyTorch's current stream (`src/solvers_torch.cpp`), which CUDA-graph capture does not record. All mHC comparisons replay CUDA graphs, so we do not compare it; we did not modify its code.

I END-TO-END TRAINING STEPS

Both runs use solver commit `7a291287` on one A100, 5 outer blocks with the method order rotated and every method in its own process; other jobs ran on the other GPUs of the host. Table 8 lists the medians over blocks and the ratios paired per block.

Table 8: Whole training steps (ms, median of 5 blocks). Official/method: paired per block, median [range]. OT: the OT loss called alone on fixed model outputs, forward and backward, ms; in parentheses, its forward call inside the step as a share of the step. *Computes a different quantity from DRAINSINKHORN: TileLang-mHC's backward is an implicit gradient at its output (table 1); DM-Count's FP32 solver transports none of the mass of cells whose kernel entries underflow.

Model	OT solver	Step	Official/method	OT
mHC GPT	PyTorch loop (official)	319	1.00	–
mHC GPT	`torch.compile`	243	1.32 [1.30, 1.58]	–
mHC GPT	DRAINSINKHORN	184	1.73 [1.71, 2.09]	–
mHC GPT	triton-sinkhorn	203	1.57 [1.56, 1.89]	–
mHC GPT	TileLang-mHC*	177	1.80 [1.79, 2.17]	–
DM-Count	DM-Count loop (official)*	455	1.00	246 (50%)
DM-Count	DRAINSINKHORN	302	1.51 [1.47, 1.55]	73 (24%)

mHC GPT. The GPT of appendix H trains from the same seed on the same FineWeb-edu tokens under every method: 4 micro-batches of 8×512 tokens per optimizer step, bf16 attention and MLP, the mHC path in FP32, fused AdamW at learning rate 10^{-3}, 23 steps (3 untimed). Only the Sinkhorn function of the model changes. The official function normalizes columns first; every other method runs its row-first map on the transposed matrix, which is the same 20-step map, except TileLang-mHC's implicit backward (section 5.2). A step is timed from its first micro-batch to the end of the optimizer step. The first losses of all methods agree within $2 \cdot 10^{-5}$; after that, bf16 training is not deterministic, and two runs of the official function differ by up to 0.027 over the 23 steps; the mean final losses of the five methods lie in 7.694–7.697.

DM-Count. DM-Count (Wang et al., 2020) (repository commit `cc5f2132`) trains a VGG-19 density regressor with an OT loss. For each 512×512 crop, it runs 100 Sinkhorn iterations with $\varepsilon = 10$ between the 64×64 normalized predicted density and uniform mass on the annotated heads, with squared pixel distances, and forms the density gradient from the potential of the density side; its `OT_Loss` loops over the images of a batch and uses a scaling-domain FP32 solver. We keep the model, the losses, and the training step of `train_helper.py` (batch 10, Adam, OT weight 0.1,

total-variation weight 0.01) and replace only the solver: one DRAINSINKHORN call per batch on the point sets padded to one size, the density side updated first as in the official code, and the loss, gradient, and logged values formed as in `OT_Loss`. The official solver starts the head-side scaling at $1/m$, which adds $\varepsilon \log m$ to the density potential; the density gradient does not depend on this constant, and we add it so that the logged objective matches. Inputs: the official QNRF weights and the 20 annotated preview images of a crowd dataset (Unidata, 2024), resized by DM-Count's preprocessing and cut into 80 seeded random crops by its training transform (0 to 2,661 heads per crop), in 8 batches. On the 79 crops with heads, the official solver ran all 100 iterations on 78 of 79. In a median 37% of the cells, all FP32 kernel entries $e^{-C_{ij}/\varepsilon}$ of the cell underflow to zero, so the official solver transports none of their mass (up to 30% of the density; more than 1% in 32 crops). Against an FP64 log-domain run of the same 100-step map, the official density potentials differ by 616–4828 per crop on the remaining cells and those of DRAINSINKHORN by at most 1.9. The two solvers therefore give different density gradients, and we compare their speed only.

J FP64 AUDIT OF RETURNED SOLUTIONS

Table 9 lists, for every method and workload in tables 2 and 4, the two-sided marginal residual of the returned potentials evaluated in FP64. For GeomLoss on `dd-t16` and `mot-win`, the audited runs used the step counts in its caption; at the counts of table 2, every timed call passed the FP32 gate (largest residuals $1.97 \cdot 10^{-3}$ and $1.66 \cdot 10^{-3}$). The residual is measured against the instance's own marginals. Both marginals of every returned solution pass the common gate of $2 \cdot 10^{-3}$ (worst $1.85 \cdot 10^{-3}$).

K REPLAY OF FLOW-MATCHING OT BATCHES

We replayed 27 batches of OT problems captured during CIFAR-10 OT flow-matching training (capture steps 0, 16, and 32; 16, 32, or 64 points per instance; 4, 8, or 16 instances per batch) with 10 paired blocks per condition. Across conditions, DRAINSINKHORN is 1.62–1.74$\times$ faster than OTT-JAX and 2.88–2.98$\times$ faster than POT; these are ranges over conditions, not confidence intervals. The replay times the OT calls only.

Table 9: FP64 two-sided ℓ_1 residuals of all returned solutions. $\leq \tau$: share of instances at most 10^{-4} (%, rounded down). DRAINSINKHORN's rows are from commit `09579cfa`, the other methods' from the run at `42566416` (same versions and inputs). GeomLoss ran 2000 annealing steps on `dd-t16` and 5000 on `mot-win` here; table 2 uses 1000 and 3000 (appendix D).

Workload	Method	Instances	$\leq \tau$	Median	p95	Max
dd-t4	DRAINSINKHORN	2304	84.9	9.56e-05	8.77e-04	1.85e-03
dd-t4	GeomLoss	2304	0.0	8.99e-04	1.36e-03	1.74e-03
dd-t4	OTT-JAX	2304	84.5	9.57e-05	9.20e-04	1.85e-03
dd-t4	POT	2304	85.1	1.08e-06	8.76e-04	1.85e-03
dd-t4	DRAINSINKHORN (grid)	2304	84.5	9.55e-05	8.76e-04	1.85e-03
dd-t4	LogSinkhornGPU	2304	85.1	1.01e-06	8.73e-04	1.85e-03
dd-t8	DRAINSINKHORN	576	99.6	9.10e-05	9.93e-05	1.00e-04
dd-t8	GeomLoss	576	0.0	9.59e-04	1.29e-03	1.44e-03
dd-t8	OTT-JAX	576	97.2	9.10e-05	9.95e-05	1.01e-04
dd-t8	POT	576	100.0	1.55e-05	5.56e-05	7.30e-05
dd-t8	DRAINSINKHORN (grid)	576	100.0	9.09e-05	9.92e-05	1.00e-04
dd-t8	LogSinkhornGPU	576	72.5	7.03e-05	2.44e-04	3.02e-04
dd-t16	DRAINSINKHORN	144	100.0	9.00e-05	9.81e-05	9.96e-05
dd-t16	GeomLoss	144	0.6	3.57e-04	6.50e-04	9.92e-04
dd-t16	OTT-JAX	144	99.3	9.04e-05	9.87e-05	1.00e-04
dd-t16	POT	144	100.0	1.10e-05	2.39e-05	5.10e-05
dd-t16	DRAINSINKHORN (grid)	144	100.0	8.98e-05	9.81e-05	1.00e-04
dd-t16	LogSinkhornGPU	144	66.6	8.41e-05	1.71e-04	3.25e-04
mot-b16	DRAINSINKHORN	3771	85.6	8.74e-05	1.13e-04	1.40e-03
mot-b16	GeomLoss	3771	62.2	2.35e-05	7.61e-04	1.70e-03
mot-b16	OTT-JAX	3771	88.4	8.80e-05	1.05e-04	1.41e-03
mot-b16	POT	3771	94.8	5.86e-06	1.03e-04	1.40e-03
mot-b32	DRAINSINKHORN	1469	98.8	8.61e-05	9.88e-05	2.65e-04
mot-b32	GeomLoss	1469	53.7	5.72e-05	6.02e-04	1.69e-03
mot-b32	OTT-JAX	1469	97.6	8.57e-05	9.87e-05	3.11e-04
mot-b32	POT	1469	99.6	4.62e-06	7.82e-06	2.64e-04
mot-b64	DRAINSINKHORN	69	97.1	8.91e-05	9.86e-05	1.00e-04
mot-b64	GeomLoss	69	37.6	8.95e-04	1.26e-03	1.40e-03
mot-b64	OTT-JAX	69	95.6	8.83e-05	9.85e-05	1.00e-04
mot-b64	POT	69	100.0	8.57e-06	3.06e-05	5.27e-05
mot-win	DRAINSINKHORN	64	98.4	8.36e-05	9.77e-05	1.00e-04
mot-win	GeomLoss	64	1.5	4.22e-04	9.01e-04	1.00e-03
mot-win	OTT-JAX	64	98.4	8.37e-05	9.75e-05	1.00e-04
mot-win	POT	64	100.0	2.93e-06	3.61e-06	7.54e-05