

GPU-Accelerated Bregman Douglas–Rachford Splitting for Discrete Optimal Transport

Yifan Xu · Shiqian Ma

October 3, 2026

Abstract We present GPU-accelerated Bregman Douglas–Rachford splitting algorithm (BDRS) for discrete optimal transport problem in three input formats: an explicit cost matrix, a point cloud with a ground cost between them, and a separable cost on a regular grid. For each input format, we propose hardware-aware designs of mathematically equivalent representations for the BDRS iterations to enhance numerical stability and empirical runtime. We benchmark the three proposed implementations against eight GPU baseline solvers from the literature on the same device. We demonstrate that our implementations of BDRS achieve state-of-the-art performance on their respective input formats. To the best of our knowledge, this is the first cross-solver study of GPU DOT solvers with a unified measure of optimality.

Keywords Optimal Transport · Bregman Douglas-Rachford Splitting · GPU Computing · First-Order Methods · Linear Programming

Mathematics Subject Classification (2020) 49Q22 · 90C05 · 65K05 · 65Y10

1 Introduction

In this paper, we consider GPU implementations of the Bregman Douglas–Rachford Splitting method (BDRS) [33,32] for solving the discrete optimal transport (DOT) problem:

Y. Xu

Department of Computational Applied Mathematics and Operations Research, Rice University, Houston, TX, USA

E-mail: yx76@rice.edu

S. Ma

Data Science and AI Institute, and Department of Applied Mathematics and Statistics, Johns Hopkins University, Baltimore, MD, USA

E-mail: sqma@jhu.edu

$$\begin{aligned}
& \min_{X \in \mathbb{R}^{m \times n}} \langle C, X \rangle \\
& \text{s.t.} \quad X \mathbf{1}_n = r, \\
& \quad \quad X^\top \mathbf{1}_m = c, \\
& \quad \quad X \geq 0,
\end{aligned} \tag{P}$$

where $C \in \mathbb{R}_+^{m \times n}$ is the cost matrix, $r \in \Delta^{m-1}$ and $c \in \Delta^{n-1}$ are the row and column marginals, and X is the transport plan. Here $\Delta^{s-1} = \{z \in \mathbb{R}^s \mid \sum_{i=1}^s z_i = 1, z \geq 0\}$ and $\mathbf{1}_n$ denotes the n -dimensional all-one vector. The dual problem of (P) is given by:

$$\begin{aligned}
& \max_{\alpha \in \mathbb{R}^m, \beta \in \mathbb{R}^n} \langle \alpha, r \rangle + \langle \beta, c \rangle \\
& \text{s.t.} \quad C - \alpha \mathbf{1}_n^\top - \mathbf{1}_m \beta^\top \geq 0.
\end{aligned} \tag{D}$$

We refer to the dual variables α and β as the row and column potentials, respectively.

The DOT problem (P) is a special linear programming problem. There exist various classical algorithms to solve the DOT problem, including network simplex [38], and specialized transportation and assignment routines such as the auction [8] and Hungarian [26, 36] algorithms. We refer to [40] for a thorough survey of computational OT. Recently, the HIO algorithm [6], a variant of the network simplex algorithm, was specialized to image grids [5] and provides exact solutions to grid DOT problems of up to 512^4 decision variables.

The cost of the aforementioned classical algorithms often scales steeply with problem size. Therefore, there has been a rising demand for scalable approximations of the solution of DOT problems in data science. As a result, a large body of recent work solves the entropy-regularized OT problem (EOT). This includes the Sinkhorn algorithm [17] and its close variants such as [2, 42, 1]. There also exist software libraries for solving EOT such as POT [24] and OTT-JAX [18]. One issue with EOT is that it requires the regularization parameter to be very small in order to well approximate the original DOT problem, which may cause numerical instability in practice.

Over the last few years, GPU implementations of first-order methods have become very competitive solvers for large-scale linear programming problems and DOT. Specifically, PDLP [3] and its GPU descendants, culminating in cuPDLPx [30], solve general LPs using only matrix-vector products, which map naturally onto GPU hardware. The same machinery has been specialized to DOT as a structured LP: PDOT [31] builds a GPU-accelerated solver that adapts restarted PDHG to the DOT problem by applying its marginalization operator matrix-free, and DROT [34] is a GPU-accelerated Douglas-Rachford splitting on the transport polytope. These solvers place no assumption on the ground cost other than its non-negativity, at the price of forming and traversing the full $m \times n$ cost and plan, whose size becomes a key memory bottleneck as the size of the problem scales up.

Existing GPU-accelerated DOT solvers can roughly be classified to three categories based on the structure of the input format of the cost, as suggested in the OTT-JAX toolbox [18]:

- *General*: the cost is provided as an explicit matrix;
- *Point Cloud*: point coordinates are provided with a ground cost between them;
- *Grid*: the marginals are provided on a shared grid, with per-axis costs.

The classical and GPU first-order methods mentioned above operate on the *General* format: the cost is held explicitly in memory as an $m \times n$ matrix .

Driven by specific applications, a parallel line of work assumes further structure in the problem. When the problem arrives as coordinates of points in Euclidean space together with a ground distance between them, as in 3D point-cloud and shape comparison [48, 49], color transfer [45], and domain adaptation [16], the problem falls in the *Point Cloud* format. Coarse-to-fine *multiscale* solvers exploit this special structure to avoid forming the full cost. Specifically, HALO [49] builds a resolution hierarchy and solves the inner subproblems using a GPU-accelerated general LP solver, and M3S [13] is a multiscale entropic Sinkhorn. In applications such as image interpolation [10], histogram-based retrieval [41], the marginals in the *Grid* format are given as histograms on a shared rectangular grid and the cost is given by *separable* per-axis terms. For this class of problems, flow reformulations such as [28] and [4] open the door to the library of network flow algorithms. Building on this, HOT [50] rewrites squared-Euclidean grid OT as a reduced row/column-flow linear program that never forms the $m \times n$ transport plan. See Section 2 for a formal definition of the formats.

Recently, Ma et al. proposed the BDRS algorithm [33] for solving the maximal monotone operator inclusion problems. When specialized to DOT ($\mathcal{P}$), the BDRS algorithm takes the following form:

$$u^k = r / ((X^{k-1} \odot K) v^{k-1}), \quad (1a)$$

$$v^k = c / ((X^{k-1} \odot K)^\top u^k), \quad (1b)$$

$$X^k = \text{diag}(u^k)(X^{k-1} \odot K) \text{diag}(v^k), \quad (1c)$$

where $K := e^{-C/\eta}$ – the componentwise exponential, i.e., $K_{ij} := e^{-C_{ij}/\eta}$, $\forall i, j$, and $\eta > 0$ is a step size. Throughout this paper, for y and z with the same dimension, $y \odot z$ denotes their componentwise multiplication and y/z denotes their componentwise division. In this paper, we consider GPU implementations of this algorithm for DOT.

Our Contributions. Our main contributions are summarized below.

- We derive a *factored representation* of the BDRS algorithm [33]. We show that the logarithm of the transport plan in the BDRS iterations can be represented as

$$\log X_{ij}^k = \xi_i^k + kL_{ij} + \zeta_j^k,$$

- where $L_{ij} := -C_{ij}/\eta$, and the vectors ξ^k, ζ^k obey a two-term recursion. Under the factored representation, the BDRS iterations can stay *matrix-free*, reducing the memory cost of the iterates from $O(mn)$ to $O(m+n)$.
- We provide *three implementations* of BDRS to solve DOT problems, each specialized for one problem format. In particular, all three implementations adopt the factored form, but differ in how the term related to cost, L_{ij} , reaches the arithmetic. Our *General* solver reads L_{ij} from a stored $m \times n$ matrix, so the cost is the only large 2D array on the device and dense problems of up to 2^{32} decision variables fit in 80 GB of VRAM, the largest that we are aware of among GPU-accelerated solvers of $(\mathcal{P})$ for this format. Our *Point Cloud* solver recomputes L_{ij} from the point coordinates in registers, so the entire solve stays matrix-free and alleviates the memory constraint. Our *Grid* solver exploits the separability of the cost to read L_{ij} directly from two 1D tables, achieving state-of-the-art runtimes while occupying just megabytes of memory at our largest scale.
 - We propose the *c-transform post-processing*, adopted by all three solvers. At termination checks, a c-transform of the running dual variables produces a feasible set of *corrected dual variables* at the cost of roughly one extra iteration. Then, the KKT residual of the transport plan together with the corrected dual variables is checked against the user-provided tolerance. On the instances we measured, stopping with the corrected dual variables meets the tolerance at an earlier check than stopping with the native ones, whose dual residual lags, and therefore accelerates the solve.
 - We assemble *eight GPU-accelerated DOT solvers* from the literature, spanning all three formats, into a *single benchmark*. Each solver runs its own native stopping rule, and we recover a primal–dual pair for the original problem $(\mathcal{P})$ from each solver’s delivered state. Since the published implementations of some baselines do not provide the dual variables, we derive the required dual mappings. Every solver is thereby scored by the same KKT residual, evaluated on their primal–dual pair of the same problem. To the best of our knowledge, this is the first cross-solver study of GPU DOT solvers with a unified measure of optimality, and the resulting accuracy–time Pareto comparison on a common GPU device constitutes our main experimental contribution.

Organization. The remainder of the paper is organized as follows. Section 2 defines the three problem formats, and states the KKT residual used throughout. Section 3 reviews the BDRS iteration for DOT and develops the c-transform post-processing used in all three BDRS solvers. Section 4 presents the factored form of the BDRS iterates and provides the implementation details of the three solvers. Section 5 reports the cross-solver benchmark and the supporting computational studies. Section 6 concludes the paper and discusses some possible future directions. Appendix A derives the primal–dual recovery for every baseline.

2 Preliminaries

The discrete OT problem. As the first step of every solve, our solvers floor the marginals internally because their log-domain updates require strict positivity. Any zero entry is raised to a floor δ_m and then the marginal is normalized with $\delta_m = 10^{-12}$ by default. We write $\tilde{r} := (r + \delta_m \mathbf{1}_m)/(1 + m\delta_m)$ and $\tilde{c} := (c + \delta_m \mathbf{1}_n)/(1 + n\delta_m)$ for the floored marginals, and use (r, c) for the instance's original marginals. The floored pair exists only for the BDRS iteration, while termination-relevant quantity is evaluated against the original (r, c) . Next, we normalize the cost matrix $C \leftarrow C/\max(C)$ to ensure $\max(C) = 1$. Since the optimal solution set of $(\mathcal{P})$ is invariant under a positive rescaling of C , this establishes a common scale against which step-sizes and residual magnitudes are read, without changing the primal solution X^* . As a linear program with a nonempty, bounded feasible set, $(\mathcal{P})$ attains its optimum and strong duality holds at optimum, so its optimal value equals that of $(\mathcal{D})$.

Problem formats. Section 1 classified prior solvers by the structure of the problem. We now define the three formats formally. In the definitions below, problem *structure* and problem *format* play distinct roles. The structure is a latent property of the problem and makes certain format of the problem possible, while the format is that possibility materialized in memory. A solver consumes formats, not structure. By exploiting problem structure, our solvers that consume more concise problem formats are able to achieve more desirable memory and time complexities.

Definition 1 (General). *In the General format, the cost $C \in \mathbb{R}_+^{m \times n}$ is a given nonnegative matrix, and the marginal distributions r and c are given probability distributions in their respective dimensions.*

Definition 2 (Point Cloud). *An instance in the Point Cloud format is specified by points $x_1, \dots, x_m \in \mathbb{R}^d$ and $y_1, \dots, y_n \in \mathbb{R}^d$, a ground cost $\kappa: \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_+$, and marginals $r \in \Delta^{m-1}$, $c \in \Delta^{n-1}$. This induces problem $(\mathcal{P})$ with the cost matrix $C_{ij} := \kappa(x_i, y_j)$, $i \in [m]$, $j \in [n]$.*

Definition 3 (Grid). *An instance of the Grid format is specified by grid dimensions $h, w \in \mathbb{N}$, axis costs $\varphi_r: \{0, 1, \dots, h-1\} \rightarrow \mathbb{R}_+$ and $\varphi_c: \{0, 1, \dots, w-1\} \rightarrow \mathbb{R}_+$, and marginals $r, c \in \Delta^{hw-1}$ on the common index set $[h] \times [w]$, where $[s] := \{1, 2, \dots, s\}$. Let $N := hw$. The provided data induces problem $(\mathcal{P})$ with $m = n = N$ and the separable cost matrix*

$$C_{(p,q),(p',q')} := \varphi_r(|p - p'|) + \varphi_c(|q - q'|),$$

indices written as row-column pairs under a fixed identification of $[h] \times [w]$ with $[N]$.

These formats nest by generality: every Grid instance we consider is also a Point Cloud instance, and every Point Cloud instance is a General instance. A solver built for a broader format therefore also handles instances from a narrower one (by converting the input data into the broader format), whereas

Table 1 Device memory requirement and per-iteration FLOPs, by format. d is the dimension of the point clouds, and typically $d \ll \min(m, n)$. s is a parameter dependent on data sparsity, with $s \leq mn$ and typically $s \ll mn$.

Format	Solver	Memory	Flops per iteration
General	General (ours)	$O(mn)$	$O(mn)$
	DROT [34]	$O(mn)$	$O(mn)$
	PDOT [31]	$O(mn)$	$O(mn)$
	cuPDLPx [30]	$O(mn)$	$O(mn)$
	Sinkhorn [17, 24]	$O(mn)$	$O(mn)$
Point Cloud	Point Cloud (ours)	$O((m+n)d)$	$O(mnd)$
	HALO [49]	$O((m+n)d+s)$	data-dependent
	M3S [13]	$O(m+n+s)$	$O(s)$
Grid	Grid (ours)	$O(N)$	$O(N(h+w))$
	HOT [50]	$O(N^{3/2})$	$O(N^{3/2})$
	Conv. Sinkhorn [45, 18]	$O(N)$	$O(N(h+w))$

a solver that assumes more structure attains cheaper iterations and/or requires less device memory. We develop one implementation of BDRS for each format. Table 1 summarizes the device memory and the per-iteration FLOPs of our three solvers against the GPU baselines of each format. The Point Cloud rows are read under the assumption that each evaluation of an entry of C (via computing $\kappa(x_i, y_j)$) costs $O(d)$ FLOPs, a common assumption every Point Cloud problem in this paper meets. Notably, in the General and Grid rows, our BDRS solvers (which target the original DOT problem ($\mathcal{P}$)) have identical memory requirements and FLOPs per iteration as the respective entropic (Sinkhorn) solvers. This is not a coincidence, and we defer further analysis and comparison of the two algorithms to Remark 2.

Remark 1. Every nonnegative C of the General format is syntactically also a Point Cloud instance: with $\kappa(u, v) = (\langle u, v \rangle)_+$, $x_i = e_i$, and y_j the j -th column of C , one has $C_{ij} = \kappa(x_i, y_j)$ with $d = m$; alternatively using the rows of C gives $d = n$, so $d = \min(m, n)$ suffices. However, this transformation does not buy more desirable memory requirements nor iteration complexity. A general $C \in \mathbb{R}^{m \times n}$ has mn independent entries, so any exact point-cloud representation must satisfy $(m+n)d \geq mn$, i.e., $d \geq mn/(m+n)$, and at that dimension the format’s $O((m+n)d)$ memory requirement and $O(mnd)$ per-iteration FLOPs meet or exceed that of the General solver’s. As a result, the Point Cloud format and its associated solver are designed for DOT instances whose cost has a low parametric dimension d , meaning that the cost C should be expressible by $O((m+n)d)$ parameters rather than $O(mn)$, with $d \ll mn/(m+n)$. In particular, $d \in \{2, 3\}$ for every Point Cloud instance in this paper.

Feasibility and Optimality. In the process of a solve, we track three residuals. The primal residual $r_P(X) = \|X\mathbf{1}_n - r\|_2 + \|X^\top \mathbf{1}_m - c\|_2 + \|\min(X, 0)\|_2$ measures the violation of the marginal and nonnegativity constraints. Since the BDRS iterates maintain $X \geq 0$ by construction, the third term vanishes for all BDRS iterates and our solvers never compute it. The dual residual

$r_D(\alpha, \beta) = \|(\alpha \oplus \beta - C)_+\|_F$ measures the violation of dual feasibility $\alpha \oplus \beta \leq C$. Here $\alpha \oplus \beta := \alpha \mathbf{1}_n^\top + \mathbf{1}_m \beta^\top \in \mathbb{R}^{m \times n}$. The duality gap of a primal–dual pair is $g(X, \alpha, \beta) := \langle C, X \rangle - (\langle \alpha, r \rangle + \langle \beta, c \rangle)$, which measures suboptimality. By weak duality, the dual objective of any feasible (α, β) lower-bounds the optimal cost, and by strong duality, the gap is zero if and only if the feasible primal–dual pair is also optimal. Therefore, a small gap evaluated at a near feasible primal–dual pair certifies near-optimality.

Error metrics and stopping criteria. All solvers are compared on the three residuals above, each evaluated in $O(m + n)$ memory so that the full $m \times n$ transport plan is never materialized if not necessary. We aggregate them into the *KKT residual* of a primal–dual triple,

$$\text{KKT}(X, \alpha, \beta) := \max\{r_P(X), r_D(\alpha, \beta), |g(X, \alpha, \beta)|\}. \quad (2)$$

All three of our solvers accepts a solution (X^k, α^k, β^k) at a termination check when $\text{KKT}(X^k, \alpha^k, \beta^k) \leq \text{tol}$, where tol is a user-specified tolerance and defaults to 10^{-4} .

3 BDRS with c-transform post-processing

3.1 The Bregman Douglas–Rachford splitting iteration

Our solvers implement the BDRS method of [33], which is the classical Douglas–Rachford splitting [21, 29, 23] with its two Euclidean proximal maps replaced by Bregman proximal maps [11, 12, 22]. We apply it to $(\mathcal{P})$ under the Boltzmann–Shannon entropy kernel $h(X) = \sum_{i,j} X_{ij}(\log X_{ij} - 1)$. The domain of h enforces the constraint $X \geq 0$ throughout the iterates, and each resolvent is a Kullback–Leibler (KL) projection onto one marginal distribution, available in closed form. The algorithm BDRS for DOT is given in (1) and we refer to [33] for details about the algorithm. The primal iterate $X^k \geq 0$ plays the role of the Lagrange multiplier of the dual inequality $\alpha \oplus \beta \leq C$, with (1c) as its multiplier update, and the scalings u^k and v^k carry the potentials of $(\mathcal{D})$ as $\alpha^k = \eta \log u^k$ and $\beta^k = \eta \log v^k$. We start from $X^0 = \frac{1}{mn} \mathbf{1}_m \mathbf{1}_n^\top$ and $v^0 = \mathbf{1}_n$, running (1) for $k = 1, 2, \dots$. The step-size η is held constant across iterations. Its deployed value is set by a fixed rule of the ground cost, given in Section 5.1. The algorithm is robust to the choice of η . See Section 5 for a study of how the solvers’ iteration count responds to changes in η .

Every factor in (1a)–(1c) is strictly positive, so $X^k > 0$ for all k in exact arithmetic. However, the guarantee diminishes in floating-point arithmetic. An entry of X^k that falls below the smallest positive representable number is flushed to numerical zero, which first destroys positivity and then finiteness. We implement (1) in the log domain to address this issue and restore numerical stability. A detailed discussion can be found in Section 4.

Remark 2. The iteration (1) can be interpreted as the alternating direction exponential multiplier method (ADEMM) [33]. More specifically, the (1a) and (1b) steps minimize, alternately over the two potential blocks, the Lagrangian of $(\mathcal{D})$ penalized by an exponential rather than quadratic penalty function (as in the classical exponential multiplier method [7, 46]), and (1c) is the corresponding multiplier update. Moreover, we draw connections between (1) and the Sinkhorn iterations, whose iteration on the entropically regularized problem with regularization ε is

$$u^k = r / (K_\varepsilon v^{k-1}), \quad v^k = c / (K_\varepsilon^\top u^k), \quad K_\varepsilon := e^{-C/\varepsilon}. \quad (3)$$

We make the observation that (1a) and (1b) are Sinkhorn updates with the constant kernel K_ε replaced by the iteration-dependent $X^{k-1} \odot K$. However, BDRS is applied to the original DOT $(\mathcal{P})$ rather than its entropically regularized surrogate, and η need not be close to 0 for a fixed point of (1) to be an exact solution to $(\mathcal{P})$. It thereby avoids the ill-conditioning that a small η causes in Sinkhorn. This algorithmic connection resolves the parity observed in Table 1. An iteration of BDRS has identical cost as compared to a Sinkhorn iteration because (1) pays only one additional matrix Hadamard product to obtain $X^{k-1} \odot K$, the cost of which can be absorbed by the remaining computation in big- O notation. In other words, an iteration whose fixed points are exact solutions of $(\mathcal{P})$ costs what an entropic iteration costs, up to constants.

3.2 The c -transform certificate and the termination check procedure

In our numerical experiments, we observed that the three optimality residuals in (2) behave differently along the trajectory generated by (1). Each u - and v -step enforces one set of marginal constraints exactly, and the primal residual r_P^k meets the tolerance tol in relatively few iterations. The native duality gap $g(X^k, \alpha^k, \beta^k)$ at the running iterates meets it within a comparable number. However, the dual residual r_D^k does not. At the termination check where the other two first meet the tolerance, r_D^k is often still orders of magnitude above them. Section 5.4 presents a numerical demonstration of this dual residual stalling phenomenon.

The stall matters because, with a greatly infeasible dual, a small duality gap is not a valid measure of suboptimality. The dual objective $\langle \alpha^k, r \rangle + \langle \beta^k, c \rangle$ lower-bounds the optimal value of $(\mathcal{P})$ only when (α^k, β^k) is feasible. When r_D^k is relatively large, the dual objective may overshoot, so the native gap $g(X^k, \alpha^k, \beta^k)$ is *optimistic*.

We propose to restore dual feasibility in a *one-shot* post-processing step. Fixing the row potential α , the largest column potential that keeps dual feasibility $\alpha \oplus \beta \leq C$ is the c -transform [40]

$$\beta'_j := \min_{1 \leq i \leq m} (C_{ij} - \alpha_i), \quad \forall j \in [n]. \quad (4)$$

By construction, we have $\alpha_i + \beta'_j \leq C_{ij}$ for every (i, j) , so (α, β') is exactly dual-feasible. Since β' is the largest feasible choice given α , it yields the tightest gap

available for that choice of α . The symmetric transform $\alpha'_i := \min_{1 \leq j \leq n} (C_{ij} - \beta_j)$ does the same for a given choice of β , and we apply the two sequentially to the running iterates (α^k, β^k) to obtain (α', β') , which we call the *corrected dual*.

Because (α', β') is feasible, weak duality makes $\langle \alpha', r \rangle + \langle \beta', c \rangle$ a true (though not necessarily tight) lower bound to the optimal objective value of $(\mathcal{P})$. When X^k is in addition near-feasible, i.e., when r_P^k meets the user-defined tolerance, we claim that the gap $g(X^k, \alpha', \beta')$ of the primal iterate and the *corrected dual variables* is a genuine measure of its suboptimality. Therefore, our solvers adopt the certified gap $g(X^k, \alpha', \beta')$ as opposed to the optimistic native gap $g(X^k, \alpha^k, \beta^k)$.

This gives the termination check procedure we implement in all three BDRS solvers. At a check we form (α', β') from (4) and its α -counterpart, and terminate at tolerance tol once $\text{KKT}(X^k, \alpha', \beta') = \max\{r_P^k, |g(X^k, \alpha', \beta')|\} < \text{tol}$. Evaluating r_P^k at a termination check is cheaper than its definition suggests. After the v -update, the column sums of X^k agree with c exactly, up to the flooring of the marginals:

$$\sum_i X_{ij}^k = v_j^k ((X^{k-1} \odot K)^\top u^k)_j = \tilde{c}_j \approx c_j \quad \forall j.$$

Therefore, all three solvers' termination checks evaluate the row residual alone. The c -transform runs only at the checks where $r_P^k < \text{tol}$. Termination checks are performed every T_{chk} iterations, with a default value dependent on problem size. Details of the T_{chk} scheme, as well as a numerical study on the robustness of our solvers with respect to T_{chk} , can be found in Section 5.

Remark 3. *The corrected duals (α', β') serve only to check termination. If the termination check fails and (X^k, α', β') is not accepted, then the iteration resumes from the un-transformed primal-dual iterate (X^k, α^k, β^k) , not from the corrected duals.*

Our three solvers share the following template of abstract operations.

1. **NORMALIZE.** Rescale the cost so that $\max(C) = 1$ and floor the marginals, yielding $(C, \tilde{r}, \tilde{c})$ as in Section 2. Also, initialize $(X^0, v^0) = (\frac{1}{mn} \mathbf{1}_m \mathbf{1}_n^\top, \mathbf{1}_n)$.
2. **BDRSSTEP.** For $k = 1, 2, \dots$, advance (X^{k-1}, v^{k-1}) to (X^k, v^k) by the BDRS iteration (1). Note that u^{k-1} is never consumed by the k -th iteration, and therefore is never carried as a part of algorithm state.
3. **PRIMALRES.** When $k \equiv 0 \pmod{T_{\text{chk}}}$, evaluate the primal residual $r_P(X^k)$. If $r_P > \text{tol}$, go back to step 2.
4. **CTransform and GAP.** Otherwise, form the corrected duals (α', β') by (4) and its α -counterpart, and compute the gap $g(X^k, \alpha', \beta')$. If $|g| < \text{tol}$, return (X^k, α', β') ; otherwise, go back to step 2.

If the iteration budget K_{max} is exhausted, the solver returns $(X^{K_{\text{max}}}, \alpha', \beta')$ with the potentials c -transformed at the final iterate, so that the delivered solution is dual feasible whether the solver accepted the iterates or not.

4 Implementation Details

In this section, we describe the implementation details of our solvers. In particular, everything described in this section stays mathematically equivalent to the BDRS with c-transform post-processing algorithm described in the previous section. We first motivate and develop the *log-domain factored form* of the BDRS iteration in Subsection 4.1, which all three solvers carry. Then, we describe the specific details of each of our solvers, from the least structured format to the most: Subsections 4.2, 4.3, and 4.4 describe our implementations of BDRS for the General, Point Cloud, and Grid formats, respectively.

4.1 The log-domain factored form

Implementing (1) as-is causes numerical issues. Recall that the standard IEEE floating-point representation [25] has a finite exponent range. The update (1c) is multiplicative and hence additive in the exponents: each iteration multiplies u^k , X^{k-1} , K and v^k , so the exponent of X_{ij}^k accumulates over all k iterations, and the factor K alone contributes $-kC_{ij}/\eta$ to it, a term unbounded below in k whenever $C_{ij} > 0$. Empirically, the entries of the transport plan that decay toward zero cross the smallest-representable FP64 threshold 5×10^{-324} within the first hundreds of iterations. There, FP64 underflows the entry to numerical zero, and a zero entry in the transport plan stays zero under (1c). Moreover, if an underflowed entry carried most mass of its row (resp. column), the scaling update (1a) (resp. (1b)) inflates towards infinity, and the c-transform as well as the duality gap become undefined.

To address these issues, all three solvers run in the *log domain*, storing logarithms of the iterates rather than the iterates themselves. In particular, an entry that underflows FP64 at 5×10^{-324} has logarithm ≈ -745 , a double well within the representable range, so the accumulation of (1c) becomes an ordinary addition of exponents.

Our next proposition shows that, in this domain, we can factor the transport plan $\log X$ into two accumulation vectors and the iteration counter.

Proposition 1 (The factored form of the log-domain BDRS iteration). *Denote $L := \log K = -C/\eta$. Let $\{(u^k, v^k, X^k)\}$ be the sequence of iterates generated by (1) with initializations $X^0 = \frac{1}{mn} \mathbf{1}_m \mathbf{1}_n^\top$, $v^0 = \mathbf{1}_n$, and define the vectors $\xi^k \in \mathbb{R}^m$ and $\zeta^k \in \mathbb{R}^n$ by*

$$\xi_i^k := -\log m + \sum_{t=1}^k \log u_i^t, \quad \zeta_j^k := -\log n + \sum_{t=1}^k \log v_j^t, \quad (5)$$

for all $i \in [m]$ and $j \in [n]$. Then, for every $k \geq 0$,

$$\log X_{ij}^k = \xi_i^k + kL_{ij} + \zeta_j^k \quad \forall (i, j) \in [m] \times [n], \quad (6)$$

and for every $k \geq 1$, starting from $\xi_i^0 = -\log m$ and $\zeta_j^0 = -\log n$, the vectors obey the recursions

$$\xi_i^k = \log r_i - \text{LSE}_j(kL_{ij} + \tilde{\beta}_j), \quad \zeta_j^k = \log c_j - \text{LSE}_i(kL_{ij} + \xi_i^k), \quad (7)$$

where $\tilde{\beta}_j := \zeta_j^{k-1} + \log v_j^{k-1}$ and $\text{LSE}_i, \text{LSE}_j$ denote the Log-Sum-Exp (LSE) operation over the index i and j of the input matrix, respectively.

Proof. First, we prove (6) by induction on k . **Base case:** at $k = 0$, $\xi_i^0 + 0 \cdot L_{ij} + \zeta_j^0 = -\log(mn) = \log X_{ij}^0$, as desired. **Inductive step:** assume that (6) holds at iteration $k-1$. Starting by taking the logarithm for both sides of (1c),

$$\begin{aligned} \log X_{ij}^k &= \log u_i^k + \log X_{ij}^{k-1} + L_{ij} + \log v_j^k \\ &= (\xi_i^{k-1} + \log u_i^k) + kL_{ij} + (\zeta_j^{k-1} + \log v_j^k) \\ &= \xi_i^k + kL_{ij} + \zeta_j^k, \end{aligned}$$

as desired.

Next, we prove (7) from (6). Using (6) at iteration $k-1$ and taking logarithms for both sides of (1a),

$$\begin{aligned} \log u_i^k &= \log r_i - \log \left(\sum_j \exp(\log X_{ij}^{k-1} + L_{ij} + \log v_j^{k-1}) \right) \\ \log u_i^k &= \log r_i - \text{LSE}_j(\xi_i^{k-1} + kL_{ij} + \zeta_j^{k-1} + \log v_j^{k-1}) \\ \log u_i^k &= \log r_i - \xi_i^{k-1} - \text{LSE}_j(kL_{ij} + \tilde{\beta}_j) \\ \xi_i^{k-1} + \log u_i^k &= \log r_i - \text{LSE}_j(kL_{ij} + \tilde{\beta}_j) \\ \xi_i^k &= \log r_i - \text{LSE}_j(kL_{ij} + \tilde{\beta}_j), \end{aligned}$$

where the second equality substitutes (6) at iteration $k-1$ into the definition of LSE, and the third moves the j -independent term ξ_i^{k-1} out of the LSE. Similarly, taking logarithms for both sides of (1b),

$$\begin{aligned} \log v_j^k &= \log c_j - \text{LSE}_i(\log X_{ij}^{k-1} + L_{ij} + \log u_i^k) \\ \log v_j^k &= \log c_j - \text{LSE}_i(\xi_i^{k-1} + \log u_i^k + kL_{ij} + \zeta_j^{k-1}) \\ \log v_j^k &= \log c_j - \zeta_j^{k-1} - \text{LSE}_i(kL_{ij} + \xi_i^k) \\ \zeta_j^{k-1} + \log v_j^k &= \log c_j - \text{LSE}_i(kL_{ij} + \xi_i^k) \\ \zeta_j^k &= \log c_j - \text{LSE}_i(kL_{ij} + \xi_i^k), \end{aligned}$$

as desired. □

Remark 4. *Proposition 1 changes the memory required for the algorithm state. By (6), the plan X^k is determined by the two vectors ξ^k, ζ^k , the counter k and the fixed matrix L , and by (7), the next pair ξ^{k+1}, ζ^{k+1} is computed from the current one, so the state a solver carries from one iteration to the next is ξ^k, ζ^k and $\log v^k$, for a total of $O(m+n)$ memory requirement. As a result, the iterates are never the cause of a memory bottleneck, whatever the format.*

The log-sum-exp operation. Both the numerical stability and the computational cost of a BDRS iteration rest on how the two LSE reductions of (7) are evaluated. We apply two standard implementation tricks together.

The first is the *max shift* [9]. Written directly, $\log \sum_i \exp(a_i)$ overflows once any a_i exceeds $\log(1.8 \times 10^{308}) \approx 710$. Subtracting the maximum away from each summand,

$$(\text{LSE } Z)_j := \max_i Z_{ij} + \log \sum_i \exp(Z_{ij} - \max_i Z_{ij}) \equiv \log \sum_i \exp(Z_{ij}), \quad (8)$$

we make every argument of $\exp$ non-positive, so each summand lies in $(0, 1]$ and the sum in $[1, m]$, inside the normal-number window with consistent relative spacing between representable numbers.

The second is the *online* form [35]. The max-shift approach (8) must already know the maximum in order to aggregate the summands, so it either reads the arguments twice, once for the maximum and once for the sum, or holds all of them at once. The online form does neither. It represents any partial reduction by a *state* (μ, s) , with μ the largest argument the partial reduction has absorbed and $s = \sum \exp(a_i - \mu)$ the sum over those arguments, so that the value of the partial reduction is $\mu + \log s$ and a single argument a is the state $(a, 1)$. Two states combine by the merge operator $\boxplus$ defined as

$$(\mu_1, s_1) \boxplus (\mu_2, s_2) := (M, s_1 e^{\mu_1 - M} + s_2 e^{\mu_2 - M}), \quad M := \max(\mu_1, \mu_2), \quad (9)$$

which is associative and commutative, so a reduction may be split across the threads of a block and reassembled in any order. Absorbing one new argument a into a state is the special case $(\mu, s) \boxplus (a, 1)$, in which one of the two exponentials is $e^0 = 1$:

$$(\mu, s) \leftarrow \begin{cases} (a, s e^{\mu - a} + 1), & a > \mu, \\ (\mu, s + e^{a - \mu}), & a \leq \mu. \end{cases} \quad (10)$$

Since μ is the running maximum, every exponent in (10) is non-positive and $s \in [1, t]$ after t arguments, so the online form keeps the guarantee of (8) while reading each argument once. We quantify the concrete effect of the difference in memory traffic in the roofline analysis below.

Roofline analysis. On top of numerical stability, the factored form also makes the iterations computationally efficient on a GPU at scale. In the roofline model [47], the binding constraint of a computational pipeline is set by its arithmetic intensity, the ratio of compute performed over the memory traffic moved. The BDRS iterations have low arithmetic intensity relative to the hardware ridge, so moving less bytes of memory will directly translate to less runtime. Under (7), the entire algorithm state is the vectors ξ^k , ζ^k and $\log v$, so no $m \times n$ array is moved between iterations. Concretely, take $m = n = N$ in FP64, so that one read or write pass over an $N \times N$ array moves $8N^2$ bytes. A direct implementation of (1) with $X^{k-1} \odot K$ held as single array reads it once per matrix–vector product, and once more, together with K , to write the update, resulting in five passes for a total of $40N^2$ bytes moved, against a multiply-add per entry in each product and three multiplies in the update, for a total of about $7N^2$ FLOPs. Under that implementation, an iteration of BDRS would yield an arithmetic intensity of about 0.2 FLOP per byte. In the General format, by contrast, the factored form reads the cost matrix C once per LSE, since the online form (10) touches each summand once and kL_{ij} is formed in registers and immediately discarded, for a total of two passes or $16N^2$ bytes. Note that if we adopted only the max-shift form (8), each LSE would read C twice for a total memory traffick of about $32N^2$ bytes instead of $16N^2$. Each entry costs one multiply to form kL_{ij} from C_{ij} , one addition for the vector term $\tilde{\beta}_j$ or ξ_i^k , and, in the $a \leq \mu$ branch of (10), a subtraction, an exponential and an addition; over the two passes that is about $10N^2$ FLOPs, for an intensity of about 0.6 FLOP per byte. This number changes drastically for the Point Cloud and Grid formats, as the cost matrix is supplied in alternative ways. We defer the details of their roofline analysis in their respective subsections. For reference, the NVIDIA A100-SXM4-80GB used in our experiments delivers 9.7 TFLOP/s in FP64 against 2.039 TB/s of memory bandwidth, placing the ridge of its roofline at about 4.8 FLOP per byte [37]. Both iterations’ arithmetic intensity lie well below the ridge value, so both are bandwidth bound and their time is set by bytes moved, of which the factored form moves 2.5 times fewer.¹

4.2 The General Solver

In the General format, the cost arrives as an $m \times n$ array C in device memory, and nothing is assumed about it beyond $C \geq 0$. This forces us to pass a $m \times n$ matrix through each of our iterations, and has two direct consequences. Firstly, for a certain large DOT instances, the matrix simply may not fit. Concretely, the matrix requires $8mn$ bytes in FP64, which is 34 GB at $m = n = 2^{16}$ and 550 GB at 2^{18} , the latter well beyond the 80 GB of a server GPU. Secondly, anytime the iterations need an entry from the cost matrix, it will always cost a read. The two design questions for this format are therefore how to keep

¹ The conclusion holds on modern data-center GPUs with full-rate FP64 arithmetic. One popular example would be the NVIDIA H100, whose ridges point is about 10.1 [14].

Algorithm 1: BDRS with c-transform for DOT in the General format

Input: cost $C \in \mathbb{R}_{\geq 0}^{m \times n}$; marginals $r \in \Delta^{m-1}$, $c \in \Delta^{n-1}$; step-size $\eta > 0$; marginal floor δ_m ; tolerance tol ; check period T_{chk} ; iteration budget K_{max}

Output: factored form of the plan (ξ, ζ, k) and exactly dual-feasible (α', β')

- 1 $C \leftarrow C / \max_{ij} C_{ij}$, $\tilde{r} \leftarrow (r + \delta_m \mathbf{1}_m) / (1 + m\delta_m)$; $\tilde{c} \leftarrow (c + \delta_m \mathbf{1}_n) / (1 + n\delta_m)$;
// NORMALIZE
- 2 $\xi \leftarrow -\log m \mathbf{1}_m$; $\zeta \leftarrow -\log n \mathbf{1}_n$; $\text{logv} \leftarrow \mathbf{0}_n$; // (X^0, v^0) in the factored form (5)
- 3 **for** $k = 1, 2, \dots, K_{\text{max}}$ **do**
- 4 $\tilde{\beta} \leftarrow \zeta + \text{logv}$; $\xi^{\text{old}} \leftarrow \xi$;
- 5 $\xi_i \leftarrow \log \tilde{r}_i - \text{LSE}_j(-kC_{ij}/\eta + \tilde{\beta}_j)$; // ξ -update of (7)
- 6 $\zeta^{\text{old}} \leftarrow \zeta$; $\zeta_j \leftarrow \log \tilde{c}_j - \text{LSE}_i(-kC_{ij}/\eta + \xi_i)$; // ζ -update of (7)
- 7 $\text{logv} \leftarrow \zeta - \zeta^{\text{old}}$;
- 8 **if** $k \bmod T_{\text{chk}} = 0$ **then**
- 9 $\rho_i \leftarrow \exp(\xi_i + \text{LSE}_j(-kC_{ij}/\eta + \zeta_j))$; // PRIMALRES
- 10 **if** $\|\rho - r\|_2 < \text{tol}$ **then**
- 11 $\alpha \leftarrow \eta(\xi - \xi^{\text{old}})$; $\beta'_j \leftarrow \min_i(C_{ij} - \alpha_i)$; $\alpha'_i \leftarrow \min_j(C_{ij} - \beta'_j)$;
// CTRANSFORM
- 12 $\text{obj} \leftarrow \sum_{ij} C_{ij} \exp(\xi_i - kC_{ij}/\eta + \zeta_j)$; // primal objective $\langle C, X^k \rangle$
- 13 **if** $|\text{obj} - \langle \alpha', r \rangle - \langle \beta', c \rangle| < \text{tol}$ **then**
- 14 **return** $(\xi, \zeta, k, \alpha', \beta')$; // factored representation of X ,
via (6)
- 15 $\alpha \leftarrow \eta(\xi - \xi^{\text{old}})$; $\beta'_j \leftarrow \min_i(C_{ij} - \alpha_i)$; $\alpha'_i \leftarrow \min_j(C_{ij} - \beta'_j)$; // exact-feasible dual for the returned iterate
- 16 **return** $(\xi, \zeta, k, \alpha', \beta')$; // budget exhausted; plan recovered via (6)

C the *only* $m \times n$ array on the device, and how to read it as few times per iteration as possible. The factored form answers both questions.

Representation. The General solver carries the factored state of Proposition 1 in the vectors $\xi^k \in \mathbb{R}^m$, $\zeta^k \in \mathbb{R}^n$ and $\log v^k \in \mathbb{R}^n$, and supplies the cost term of the recursions (7) by reading C_{ij} from the stored array and forming $kL_{ij} = -kC_{ij}/\eta$ in registers, one multiply per entry, at the moment the LSE consumes it. Neither K , nor L , nor any iterate is stored as an $m \times n$ array, so C is the only $m \times n$ array on the device. Each iteration streams C exactly twice, once per LSE of (7), which results in the $16N^2$ bytes of memory traffic and 0.6 FLOP per byte arithmetic intensity of Section 4.1, making this solver bandwidth bound. The plan is delivered in the factored form (ξ, ζ, k) and recovered through (6) only when a dense X is requested. In particular, our entire termination check procedure do not require the materialization of the transport plan.

Initialization and termination. The remaining steps of the template of Section 3.2 run on the same factored form. NORMALIZE takes the maximum of C in one pass and rescales C . The marginals are floored by adding $\delta_m \mathbf{1}$ and renormalizing, giving the $(\tilde{r}, \tilde{c})$ the iteration consumes; and the initial iterates $(X^0, v^0) = (\frac{1}{mn} \mathbf{1}_m \mathbf{1}_n^\top, \mathbf{1}_n)$ map to $\xi = -(\log m) \mathbf{1}_m$, $\zeta = -(\log n) \mathbf{1}_n$ and

$\log v = \mathbf{0}_n$ by (5). PRIMALRES uses the row marginal error as its proxy and requires one more row LSE over C (line 9 of Algorithm 1), and the full termination check proceeds only if $\|\rho - r\|_2 < \text{tol}$. CTRANSFORM is two reads over C that maintain two running minima arrays in $O(m+n)$ workspace, a column-wise one for $\beta'_j = \min_i (C_{ij} - \alpha_i)$ and a row-wise one for $\alpha'_i = \min_j (C_{ij} - \beta'_j)$, with $\alpha = \eta(\xi - \xi^{\text{old}})$. Lastly, the computational costly term of GAP is computing the primal objective $\langle C, X^k \rangle = \sum_{ij} C_{ij} \exp(\xi_i - kC_{ij}/\eta + \zeta_j)$, which corresponds to one read over C in chunks. A check that passes the proxy therefore reads C four times, the traffic of two iterations, and a check that fails it reads C once.

GPU implementation. The main loop of Algorithm 1 is implemented as three CUDA kernels and a counter increment, for a total of four graph nodes, captured once per solve into a CUDA graph and replayed once per iteration. The ξ -update of line 5 assigns one thread block of 256 threads to each row i . We store C in the row-major layout where the row is contiguous, so the threads of a warp read adjacent addresses. For each thread, it strides along the row, forms $-kC_{ij}/\eta + \beta_j$ in registers from the entry it has just read, with k taken from a device-resident counter, and aggregates it into its own online-LSE state (10). The 256 states are aggregated in shared memory by (9). One thread then computes $\xi^{\text{new}} = \log \tilde{r}_i - (\mu \log s)$, stores the difference $\log u_i = \xi_i^{\text{new}} - \xi_i$, which the c-transform will need later, and then sets $\xi_i \leftarrow \xi^{\text{new}}$. The ζ -update of line 6 runs against the row-major layout, and requires a more carefully treatment. Our implementation runs in two stages. In the first stage, each thread owns one column over a chunk of 1024 rows (128 when $m \leq 4096$) and keeps one online state; a block's threads take adjacent columns and move down the rows in lockstep, so every read is again coalesced, and the partial states of all chunks are written to two scratch arrays with one entry per chunk and column. In the second stage, one thread per column merges its chunk states by (9) into (μ, s) , computes $\zeta_j^{\text{new}} = \log \tilde{c}_j - (\mu + \log s)$, stores $\log v_j = \zeta_j^{\text{new}} - \zeta_j$, and sets $\zeta_j \leftarrow \zeta_j^{\text{new}}$. The termination checks run as chunked passes over C with PyTorch's native kernels [39]. The problem data C, r, c are staged from the host once, while the iteration, checks and delivery execute on the device, and at a check only scalar verdicts return to the host.

4.3 The Point Cloud Solver

The Point Cloud solver consumes a ground cost function κ on two point clouds $P = (p_1, \dots, p_m)$ and $Q = (q_1, \dots, q_n)$ in $\mathbb{R}^d$, with the (normalized) cost matrix defined as

$$C_{ij} := \kappa(p_i, q_j) / C_{\max},$$

where $C_{\max} := \max_{p \in P, q \in Q} \kappa(p, q)$ is the maximum cost between the two clouds. In particular, the cost matrix is given implicitly and compactly, and we take extra care to not materialize the full $m \times n$ cost matrix at any point in the Point Cloud solve.

Algorithm 2: BDRS with c-transform for DOT in the Point Cloud format

Input: Point clouds $P \in \mathbb{R}^{m \times d}$, $Q \in \mathbb{R}^{n \times d}$ with ground cost κ ; marginals $r \in \Delta^{m-1}$, $c \in \Delta^{n-1}$; step-size $\eta > 0$; marginal floor δ_m ; tolerance tol ; check period T_{chk} ; budget K_{max}

Output: Factored form of plan (ξ, ζ, k) and exactly dual-feasible (α', β')

```

1  $C_{\text{max}} \leftarrow \max_{i,j} \kappa(p_i, q_j)$ ;  $\tilde{r} \leftarrow (r + \delta_m \mathbf{1}_m)/(1 + m\delta_m)$ ;  $\tilde{c} \leftarrow (c + \delta_m \mathbf{1}_n)/(1 + n\delta_m)$ ;
   // NORMALIZE; costs enter as  $\kappa(p_i, q_j)/C_{\text{max}} \leq 1$ 
2  $\xi \leftarrow -(\log m)\mathbf{1}_m$ ;  $\zeta \leftarrow -(\log n)\mathbf{1}_n$ ;  $\text{logv} \leftarrow \mathbf{0}_n$ ;
3 for  $k = 1, 2, \dots, K_{\text{max}}$  do
4    $\tilde{\beta} \leftarrow \zeta + \text{logv}$ ;  $\xi^{\text{old}} \leftarrow \xi$ ;
5    $\xi_i \leftarrow \log \tilde{r}_i - \text{LSE}_j(-k\kappa(p_i, q_j)/(C_{\text{max}}\eta) + \tilde{\beta}_j)$ ; //  $\xi$ -update of (7)
6    $\zeta^{\text{old}} \leftarrow \zeta$ ;  $\zeta_j \leftarrow \log \tilde{c}_j - \text{LSE}_i(-k\kappa(p_i, q_j)/(C_{\text{max}}\eta) + \xi_i)$ ; //  $\zeta$ -update of (7)
7    $\text{logv} \leftarrow \zeta - \zeta^{\text{old}}$ ;
8   if  $k \bmod T_{\text{chk}} = 0$  then
9      $\rho_i \leftarrow \exp(\xi_i + \text{LSE}_j(-k\kappa(p_i, q_j)/(C_{\text{max}}\eta) + \zeta_j))$ ; // PRIMALRES
10    if  $\|\rho - r\|_2 < \text{tol}$  then
11       $\alpha \leftarrow \eta(\xi - \xi^{\text{old}})$ ;  $\beta'_j \leftarrow \min_i (\kappa(p_i, q_j)/C_{\text{max}} - \alpha_i)$ ;  $\alpha'_i \leftarrow$ 
         $\min_j (\kappa(p_i, q_j)/C_{\text{max}} - \beta'_j)$ ; // CTRANSFORM
12       $\text{obj} \leftarrow \sum_{i,j} \frac{\kappa(p_i, q_j)}{C_{\text{max}}} \exp(\xi_i - k\kappa(p_i, q_j)/(C_{\text{max}}\eta) + \zeta_j)$ ; // primal
        objective  $\langle C, X^k \rangle$ 
13      if  $|\text{obj} - \langle \alpha', r \rangle - \langle \beta', c \rangle| < \text{tol}$  then
14        return  $(\xi, \zeta, k, \alpha', \beta')$ ; // factored representation of  $X$ ,
        recoverable via (6)
15  $\alpha \leftarrow \eta(\xi - \xi^{\text{old}})$ ;  $\beta'_j \leftarrow \min_i (\kappa(p_i, q_j)/C_{\text{max}} - \alpha_i)$ ;
     $\alpha'_i \leftarrow \min_j (\kappa(p_i, q_j)/C_{\text{max}} - \beta'_j)$ ; // feasible duals for the returned iterate
16 return  $(\xi, \zeta, k, \alpha', \beta')$ ; // budget exhausted, return final iterate
```

Representation. The Point Cloud solver carries the same factored state of Proposition 1 as the other two solvers, in the vectors $\xi^k \in \mathbb{R}^m$, $\zeta^k \in \mathbb{R}^n$ and $\text{logv} \in \mathbb{R}^n$. The transport plan is recoverable via (6) upon final delivery, while the term related to the cost matrix $L_{ij} = -\kappa(p_i, q_j)/(C_{\text{max}}\eta)$ is recomputed from the coordinates in registers each time it is needed. This implies that the algorithm state occupies $O(m+n)$ memory, while the overall device memory requirement is in the order of $O((m+n)d)$ due to the storage required to store the coordinates of the point clouds.

Initialization and termination. In the Point Cloud format, NORMALIZE is a one-time pairwise maximum C_{max} defined above, computable in $O(mnd)$ FLOPs. The marginal distribution flooring procedure is identical to the one in the General solver. The initial iterates (X^0, v^0) map to the representation as $\xi^0 = -(\log m)\mathbf{1}_m$ and $\zeta^0 = -(\log n)\mathbf{1}_n$, by (5), and $\text{logv} = \mathbf{0}_n$, in $O(m+n)$. PRIMALRES, for which we use the row marginal error as a proxy, requires a row-LSE in $O(mnd)$ FLOPs. Next, with $\alpha^k = \eta \log u^k = \eta(\xi^k - \xi^{k-1})$, the CTRANSFORM simply sweeps $\beta'_j = \min_i (C_{ij} - \alpha_i^k)$ and $\alpha'_i = \min_j (C_{ij} - \beta'_j)$ while recomputing C_{ij} from coordinates, in $O(mnd)$ FLOPs using $O(m+n)$ workspace. Lastly, the primal objective, which is the

compute-intensive part of GAP, is computed via exponentiating and aggregating $\xi_i - k\kappa(p_i, q_j)/(C_{\max}\eta) + \zeta_j$ across all pairs of points while weighting them with the cost $\kappa(p_i, q_j)/C_{\max}$, again recomputed from the coordinates in $O(mnd)$ FLOPs. A check that passes the proxy therefore costs four $O(mnd)$ passes, the arithmetic of two iterations, while a check that fails the proxy costs one.

GPU implementation. Algorithm 2 is a description of our Point Cloud solver, which implements the template of Section 3.2 for the Point Cloud format. Both row and column LSEs in (7) are aggregations of mn terms, and their respective kernels assign one thread block per output entry. For the ξ -update, the block of row i loads the point p_i once, and its 256 threads iterate through the n opposing points. Each thread recomputes $\kappa(p_i, q_j)$ from the coordinates in registers, forms the term $-k\kappa(p_i, q_j)/(C_{\max}\eta) + \tilde{\beta}_j$, and updates its private online-LSE state (10). The 256 states in the thread block are then merged in shared memory by (9) and the ζ -update kernel is constructed similarly, over columns. Since both updates stream the opposing cloud as one contiguous array of points, there is no layout asymmetry and hence no special treatment is needed for the column LSE, unlike its General counterpart. The point cloud coordinates are staged from the host onto the device once, and at termination checks only scalar verdicts return to the host. As a result, this traffic does not limit the wall-clock time. An iteration launches three kernels, the ξ -update, the ζ -update, and a counter advance that increments the device-side iteration count k ; the three are captured once per solve as a CUDA graph and replayed each iteration.

Remark 5. *In the Point Cloud solver, the arithmetic is $O(mnd)$ FLOPs per update against $O((m+n)d)$ bytes moved, making the arithmetic intensity of Point Cloud kernels grow like $O(mn/(m+n))$ with problem size and the kernels are compute bound even on small DOT instances. In the accounting of Section 4.1, the cost entries are recomputed rather than read. Evaluating $\kappa(p_i, q_j)$ from the coordinates takes about $3d$ operations and replace the read, while the multiple that scales it by $-k/(C_{\max}\eta)$, the addition of the vector term and the three operations of the LSE update are as before, for a total of about $3d+4$ per entry per pass, or $(6d+8)N^2$ FLOPs per iteration with $m = n = N$. Note that the opposing point cloud occupies $8Nd$ bytes, which is at most about a megabyte at the sizes we run (with $d \in \{2, 3\}$), so it fits in the 40 MB L2 cache [15]² and need not be read from HBM. Each pass moves the two clouds, the vector term and the output once for a total of $8N(2d+2)$ bytes, or $16N(2d+2)$ per iteration. The estimated arithmetic intensity is therefore about $0.2N$ FLOP per byte at $d \in \{2, 3\}$, crossing the A100-SXM4-80GB ridge at $N = 24$ and growing linearly in N . Note that the count treats the exponential as one operation, while on the device it typically consists of a sequence of basic*

² The size of the L2 cache may vary between devices, but the point cloud coordinates should generally fit in the L2 cache of common server GPUs.

Algorithm 3: BDRS with c-transform for DOT in the Grid format

Input: axis costs φ_r, φ_c ; marginals $r, c \in \Delta^{N-1}$ on an $h \times w$ grid; step-size $\eta > 0$; marginal floor δ_m ; tolerance tol ; check period T_{chk} ; budget K_{max}

Output: Factored form of plan (ξ, ζ, k) and exactly dual-feasible (α', β')

```

1  $C_{\text{max}} \leftarrow \max_{\delta_r} \varphi_r(\delta_r) + \max_{\delta_c} \varphi_c(\delta_c)$ ;
2  $\ell_r(\delta) \leftarrow -\varphi_r(\delta)/(C_{\text{max}}\eta)$ ;  $\ell_c(\delta) \leftarrow -\varphi_c(\delta)/(C_{\text{max}}\eta)$ ; // axis tables
3  $\tilde{r} \leftarrow (r + \delta_m \mathbf{1}_N)/(1 + N\delta_m)$ ;  $\tilde{c} \leftarrow (c + \delta_m \mathbf{1}_N)/(1 + N\delta_m)$ ; // NORMALIZE: marginal floor
4  $\xi \leftarrow -(\log N)\mathbf{1}_N$ ;  $\zeta \leftarrow -(\log N)\mathbf{1}_N$ ;  $\text{logv} \leftarrow \mathbf{0}_N$ ;
5 for  $k = 1, 2, \dots, K_{\text{max}}$  do
6    $\tilde{\beta} \leftarrow \zeta + \text{logv}$ ;  $\xi^{\text{old}} \leftarrow \xi$ ;
7    $\log S_{i_r, j_c} \leftarrow \text{LSE}_{j_r}(k\ell_r(|i_r - j_r|) + \tilde{\beta}_{j_r, j_c})$ ; //  $\xi$ -update inner pass
8    $\xi_{i_r, i_c} \leftarrow \log \tilde{r}_{i_r, i_c} - \text{LSE}_{j_c}(k\ell_c(|i_c - j_c|) + \log S_{i_r, j_c})$ ; // outer pass
9    $\log T_{j_r, i_c} \leftarrow \text{LSE}_{i_r}(k\ell_r(|i_r - j_r|) + \xi_{i_r, i_c})$ ; //  $\zeta$ -update inner pass
10   $\zeta^{\text{old}} \leftarrow \zeta$ ;  $\zeta_{j_r, j_c} \leftarrow \log \tilde{c}_{j_r, j_c} - \text{LSE}_{i_c}(k\ell_c(|i_c - j_c|) + \log T_{j_r, i_c})$ ; // outer pass
11   $\text{logv} \leftarrow \zeta - \zeta^{\text{old}}$ ;
12  if  $k \bmod T_{\text{chk}} = 0$  then
13     $\log S_{i_r, j_c} \leftarrow \text{LSE}_{j_r}(k\ell_r(|i_r - j_r|) + \zeta_{j_r, j_c})$ ; // PRIMALRES inner pass
14     $\rho_{i_r, i_c} \leftarrow \exp(\xi_{i_r, i_c} + \text{LSE}_{j_c}(k\ell_c(|i_c - j_c|) + \log S_{i_r, j_c}))$ ; // outer pass
15    if  $\|\rho - r\|_2 < \text{tol}$  then
16       $\alpha \leftarrow \eta(\xi - \xi^{\text{old}})$ ; // native dual  $\alpha^k = \eta \log u^k$ 
17       $\tau_{j_r, i_c} \leftarrow \min_{i_r}(\varphi_r(|i_r - j_r|)/C_{\text{max}} - \alpha_{i_r, i_c})$ ; // CTRANSFORM  $\beta'$  inner pass
18       $\beta'_{j_r, j_c} \leftarrow \min_{i_c}(\varphi_c(|i_c - j_c|)/C_{\text{max}} + \tau_{j_r, i_c})$ ; //  $\beta'$  outer pass
19       $\tau'_{i_r, j_c} \leftarrow \min_{j_r}(\varphi_r(|i_r - j_r|)/C_{\text{max}} - \beta'_{j_r, j_c})$ ; //  $\alpha'$  inner pass
20       $\alpha'_{i_r, i_c} \leftarrow \min_{j_c}(\varphi_c(|i_c - j_c|)/C_{\text{max}} + \tau'_{i_r, j_c})$ ; //  $\alpha'$  outer pass
21       $\log S'_{i_c, j_r} \leftarrow \text{LSE}_{j_c}(k\ell_c(|i_c - j_c|) + \zeta_{j_r, j_c})$ ; // obj inner pass
22       $R^r_{i_r, j_r} \leftarrow \exp(k\ell_r(|i_r - j_r|) + \text{LSE}_{i_c}(\xi_{i_r, i_c} + \log S'_{i_c, j_r}))$ ; // row-axis marginal of  $X^k$ 
23       $R^c_{i_c, j_c} \leftarrow \exp(k\ell_c(|i_c - j_c|) + \text{LSE}_{i_r}(\xi_{i_r, i_c} + \log S_{i_r, j_c}))$ ;
24      // column-axis marginal; reuses  $\log S$ 
25       $\text{obj} \leftarrow \sum_{i_r, j_r} \varphi_r(|i_r - j_r|) R^r_{i_r, j_r} / C_{\text{max}} + \sum_{i_c, j_c} \varphi_c(|i_c - j_c|) R^c_{i_c, j_c} / C_{\text{max}}$ ;
26      if  $|\text{obj} - \langle \alpha', r \rangle - \langle \beta', c \rangle| < \text{tol}$  then return  $(\xi, \zeta, k, \alpha', \beta')$ ;
27 return  $(\xi, \zeta, k, \alpha', \beta')$ ; // budget exhausted, return final iterate

```

arithmetic operations, so the true arithmetic intensity is higher still, and our estimate is conservative.

4.4 The Grid Solver

The Grid solver consumes the problem $(\mathcal{P})$ in the Grid format of Definition 3, and is best understood as the Point Cloud solver under two further assumptions.

1. The two point clouds coincide with the same $h \times w$ grid, so that $m = n = hw =: N$, and a point p_i (resp. q_j) is identified by its row and column indices (i_r, i_c) (resp. (j_r, j_c)).
2. The ground cost is *separable* with translation-invariant axis costs, as in Definition 3. In other words, $\kappa(p_i, q_j)$ can be written as $\varphi_r(|i_r - j_r|) + \varphi_c(|i_c - j_c|)$. For example, both $\kappa_1(p_i, q_j) := \ell_1(p_i - q_j)$ and $\kappa_2(p_i, q_j) := \ell_2^2(p_i - q_j)$ are separable, with $\varphi_{r,1}(\delta) = \varphi_{c,1}(\delta) = \delta$ and $\varphi_{r,2}(\delta) = \varphi_{c,2}(\delta) = \delta^2$, respectively.

Representation. The Grid solver adopts the same factored form of Proposition 1 and stores the algorithm state in the vectors $\xi^k, \zeta^k, \log \mathbf{v} \in \mathbb{R}^N$. The main difference in representation lies in how we obtain the entries of L , the term related to the cost matrix C . In particular, from the definition of L (in Proposition 1),

$$\begin{aligned} L_{ij} &= -C_{ij}/\eta \\ &= -\kappa(p_i, q_j)/(\eta C_{\max}) \\ &= -(\varphi_r(|i_r - j_r|) + \varphi_c(|i_c - j_c|))/(\eta C_{\max}) \\ &= \ell_r(|i_r - j_r|) + \ell_c(|i_c - j_c|), \end{aligned}$$

where $\ell_r : \{0, 1, \dots, h-1\} \rightarrow \mathbb{R}_{\leq 0}$ is defined as

$$\ell_r(\delta) := -\frac{1}{\eta C_{\max}} \varphi_r(\delta),$$

and ℓ_c is defined similarly on $\{0, 1, \dots, w-1\}$. Therefore, instead of recomputing $L_{ij} = -\kappa(p_i, q_j)/(\eta C_{\max})$ from the coordinates as seen in the Point Cloud case, we can alternatively pre-compute and store ℓ_r and ℓ_c as two one-dimensional tables in device memory. Moreover, the normalizer is

$$C_{\max} = \max_{\delta_r} \varphi_r(\delta_r) + \max_{\delta_c} \varphi_c(\delta_c), \quad (11)$$

which costs at most an $O(h+w)$ scan over the two tables. For the case where axis costs are nondecreasing (e.g., ℓ_1 and ℓ_2^2 costs), it reduces to the corner-to-corner displacement $\varphi_r(h-1) + \varphi_c(w-1)$. As a result, the device memory requirement is $O(N)$. Similar to the Point Cloud solver, the plan is delivered in the factored form (ξ, ζ, k) at termination and recovered via (6) if needed.

The iteration evaluates the recursions (7) on this representation through a separable factorization of each LSE along the two axes of the grid. In particular, for the ξ -update, we aggregate across the target points $j = (j_r, j_c) \in [h] \times [w]$:

$$\text{LSE}_j(kL_{ij} + \tilde{\beta}_j) = \text{LSE}_{j_c} \left(k\ell_c(|i_c - j_c|) + \underbrace{\text{LSE}_{j_r}(k\ell_r(|i_r - j_r|) + \tilde{\beta}_{j_r, j_c})}_{\log S_{i_r, j_c}} \right), \quad (12)$$

which is an inner pass over rows, producing the intermediate table $\log S$, and an outer pass over columns, and likewise for the ζ -update. The factorization

of the LSE operation comes from the fact that the sum over j is really the double sum over (j_r, j_c) , where the column terms are constant with respect to j_r . Each update is $2N$ one-dimensional LSEs, so an iteration costs $O(N(h+w))$ FLOPs; on a square grid ($h = w = n_g$) this is $O(n_g^3) = O(N^{3/2})$, a factor $\sqrt{N}$ below the Point Cloud and General solvers, and the source of the Grid solver's wallclock speed.

Initialization and termination. In the Grid solver, each initialization and termination check operation costs no more than $O(N(h+w))$ FLOPs. NORMALIZE can be performed in $O(h+w)$, and in $O(1)$ for nondecreasing axis costs. Building the axis tables ℓ_r, ℓ_c is $O(h+w)$; the marginal flooring is identical to the previous two formats in $O(N)$. The initial iterates (X^0, v^0) map to $\xi = \zeta = -(\log N)\mathbf{1}_N$ and $\log v = \mathbf{0}_N$, as seen in the Point Cloud solver, in $O(N)$. PRIMALRES evaluates the row marginals by reusing (12). In particular, by (6),

$$\sum_j X_{ij}^k = \exp(\xi_i^k + \text{LSE}_j(kL_{ij} + \zeta_j^k)),$$

which is exactly (12) with $\tilde{\beta}$ replaced by ζ . CTRANSFORM factors along the two axes exactly as (12) does, because constant terms can be moved outside of the minimization just like how they could be moved outside of the LSE operation. Similar to (12), the c-transform $\beta'_j = \min_i (C_{ij} - \alpha_i)$ over the source points $i = (i_r, i_c) \in [h] \times [w]$ becomes the double minimum over (i_r, i_c) , in which the column term $\varphi_c(|i_c - j_c|)/C_{\max}$ is constant with respect to i_r and moves outside the inner minimization:

$$\beta'_{j_r, j_c} = \min_{i_c} \left(\varphi_c(|i_c - j_c|)/C_{\max} + \min_{i_r} \left(\varphi_r(|i_r - j_r|)/C_{\max} - \alpha_{i_r, i_c} \right) \right).$$

This is two nested minimizations over the two axes, computable in $O(N(h+w))$ FLOPs. A similar argument can be made for the c-transform in the other direction, to obtain α' . The primal objective, the dominant term of GAP, factors along the two axes as well:

$$\begin{aligned} \langle C, X^k \rangle &= \frac{1}{C_{\max}} \sum_{ij} \varphi_r(|i_r - j_r|) X_{ij}^k + \frac{1}{C_{\max}} \sum_{ij} \varphi_c(|i_c - j_c|) X_{ij}^k \\ &= \frac{1}{C_{\max}} \sum_{i_r, j_r} \varphi_r(|i_r - j_r|) \underbrace{\sum_{i_c, j_c} X_{ij}^k}_{=: R_{i_r, j_r}^r} + \frac{1}{C_{\max}} \sum_{i_c, j_c} \varphi_c(|i_c - j_c|) \underbrace{\sum_{i_r, j_r} X_{ij}^k}_{=: R_{i_c, j_c}^c}, \end{aligned}$$

since each cost φ is constant with respect to the axis along which we are aggregating. Substituting the factored plan (6) with $L_{ij} = \ell_r(|i_r - j_r|) + \ell_c(|i_c - j_c|)$,

$$\begin{aligned} R_{i_r, j_r}^r &= \sum_{i_c} \sum_{j_c} \exp(\xi_{i_r, i_c} + k\ell_r(|i_r - j_r|) + k\ell_c(|i_c - j_c|) + \zeta_{j_r, j_c}) \\ &= \exp\left(k\ell_r(|i_r - j_r|) + \text{LSE}_{i_c}(\xi_{i_r, i_c} + \text{LSE}_{j_c}(k\ell_c(|i_c - j_c|) + \zeta_{j_r, j_c}))\right). \end{aligned}$$

A similar scheme can be derived for R^c . In total, the primal objective costs three new axis-length passes, in $O(N(h + w))$ FLOPs. A check that passes the proxy therefore costs nine axis-length passes which is a little over two iterations, and one that fails the proxy costs the same as about half of an iteration.

GPU implementation. Algorithm 3 is a description of our Grid solver, which implements the template of Section 3.2 for the Grid format. Our kernels specialize to the square grids of the benchmark dataset, $h = w = n_g$ with $\varphi_r = \varphi_c$, collapsing the two axis tables to a single table ℓ . Each update is one fused kernel with a thread block per grid row. For the ξ -update, the thread block of row i_r owns the i_r -th row of the intermediate table $\log S$, which is computed once and reused by all n_g outputs of the row. The block stages the table $k\ell$ and this intermediate row in shared memory, allowing fast access during both LSE passes. The ζ -update kernel is implemented similarly. Within the block, each of the n_g outputs is computed by W threads that cooperate on its LSE operation ($W = 32, 16, 8$ at $n_g = 32, 64, 128$, so that $n_g W = 1024$ fills the block; from $n_g = 256$ onwards, we set $W = 1$ and the block is n_g threads), each thread updating its own private online-LSE state (10). Finally, the per-thread states are merged by (9) in shared memory. An iteration launches three kernels, the ξ -update, the ζ -update, and a counter advance, which are captured once per solve as a CUDA graph and replayed.

Remark 6. *In the accounting of Section 4.1, an update of the Grid solver evaluates $2n_g^3$ summands. The intermediate table $\log S$ has $N = n_g^2$ entries, each an LSE over n_g terms, and the N outputs are each an LSE over n_g terms. Each summand costs one addition to combine the table entry $k\ell$ with the vector or intermediate term, and three more for the LSE update (10) for a total of four operations. That is $8n_g^3$ operations per update, and in total, $16n_g^3$ FLOPs per iteration. On the memory side, the ξ -update reads $\tilde{\beta}$, $\log \tilde{r}$ and the old ξ and writes ξ and $\log u$; the ζ -update reads ξ , $\log \tilde{c}$ and the old ζ and writes ζ , $\log v$ and the next $\tilde{\beta}$. This amounts to eleven passes over $N = n_g^2$ elements, or $88n_g^2$ bytes per iteration.³ The arithmetic intensity is therefore at least $16n_g^3/(88n_g^2) \approx 0.2n_g$ FLOP per byte, above the 4.8 ridge from $n_g = 32$ onwards.*

At smaller scales, the absolute scale of the arithmetic is small enough such that its runtime on our device is comparable to that of a kernel launch. Concretely, at $n_g = 128$, an iteration costs about 3×10^7 FLOPs, which is a few microseconds at the device's peak rate [37] and at the scale of a kernel launch. So, up to $n_g \approx 128$, the iteration cannot fill the device and launch latency governs. This makes the CUDA graph an essential part of the Grid solver for small scale instances. For larger instances, the compute time overshadows the launch times, and the kernels become compute bound.

³ This is assuming all vectors are stored in HBM. This is a pessimistic estimate, since these N -dimensional vectors occupy $56n_g^2$ bytes and typically fit in the device L2 cache.

Table 2 Software versions recorded on the machine that experiments in Section 5 are performed on.

Component	Version
<i>Our solvers</i>	
Operating system	Linux-5.14.0-570.58.1.el9_6.x86_64-x86_64-with-glibc2.34
Python	3.11.16
PyTorch	2.13.0+cu130
CUDA (PyTorch runtime)	13.0
CUDA toolkit (<code>nvcc</code>)	Build cuda_13.0.r13.0/compiler.36424714_0
NumPy	2.4.6
SciPy	1.17.1
<i>Baselines</i>	
POT (log-domain Sinkhorn)	0.9.7
OTT-JAX (Grid Sinkhorn)	0.6.0
JAX	0.10.2
Julia (PDOT)	1.11.9
CUDA.jl (PDOT)	5.11.3
DROT	6ad8d86589 (our snapshot)
HALO	1671ae6905 (upstream)
HOT	6ad8d86589 (our snapshot)
M3S	ec100cb67e (upstream)
PDOT	19760e6cc1 (our snapshot)
cuPDL Px	931c94c0c3 (upstream)

5 Numerical Experiments

In this section, we conduct numerical experiments to

1. quantify the difference in computational speed between the three BDRS solvers, across the three formats;
2. compare the three BDRS solvers against the GPU-accelerated baselines in their respective formats;
3. quantify how the c-transform post-processing accelerates the solve for the Douglas–Rachford family of solvers, including ours and two baselines, and
4. measure how the iteration count of BDRS responds when the step size η is perturbed around its deployed value.

Table 2 lists information about the software stack that can be used to reproduce the experiments in this section.

5.1 Experiment setup

Datasets. Three datasets are used in this section, one for each DOT format. Note that for a dataset of DOT instances with more structure (e.g. Grid), we can *lift* it into its less structured counterpart by writing out the cost function κ (Point Cloud) or computing and storing the explicit C (General).

DOTmark [43] natively supplies Grid and Point Cloud instances, some of which are further lifted to General instances. It contains ten distinct classes of

images, each class holding ten grayscale square images. The dataset supplies these images at various resolutions $n_g \in \{32, 64, 128, 256, 512\}$, so that each image can be represented as a $n_g \times n_g$ matrix of nonnegative values. Similar to [49], we derive a $n_g = 1024$ tier by corner-anchored bilinear upsampling of the supplied $n_g = 512$ images. Next, within a class, a Grid DOT instance can be constructed for each pair of images of equal resolution, resulting in $\binom{10}{2} = 45$ distinct DOT instances for each class and 450 for each resolution and cost. The benchmark of Sections 5.2 and 5.3 uses all of them, and every aggregate there is over the instances a cell’s solvers co-solved; the two studies of Section 5.4 use a fixed subset of five pairs per class. To construct the DOTmark instances in Grid or Point Cloud format, we first normalize the pixel values of both images to a probability distribution over the $n_g \times n_g$ grid, so that $N = n_g^2$. Then, to obtain Grid (resp. Point Cloud) instances, we select either the ℓ_1 or ℓ_2^2 (resp. The ℓ_∞ or the unsquared ℓ_2) cost as the cost between grid coordinates. We lift the Grid DOTmark instances into Point Cloud DOTmark instances, and then lift all Point Cloud DOTmark instances into General DOTmark instances, so that benchmarking experiments can be run on any selection of (cost, format) pair.

ModelNet10 [48] natively supplies the Point Cloud instances. It is a library of computer-aided-design (CAD) 3-dimensional models of common indoor objects, distributed as triangle meshes and grouped into ten categories. From each category we take the first three shapes of its test split and pair them within the category, so that the two clouds of an instance are always distinct objects of the same kind. This gives $\binom{3}{2} \cdot 10 = 30$ pairs of objects across the dataset. Each CAD mesh is sampled area-weighted into a surface point cloud, which is then scaled isotropically into $[0, 1]^3$. For each object, $N_{\max} = 2^{19}$ samples were drawn *once* throughout our experiments. Then, we randomly fixed an ordering of these samples so that taking the first $N \leq N_{\max}$ points always give a nested, reproducible subsample that’s individually unbiased. Moreover, we select two notions of cost between the same point clouds: the squared Euclidean cost and the unsquared ℓ_2 cost.

Lastly, we used **random dense costs** as way to generate natively General format DOT instances. In particular, each entry of the cost matrix C_{ij} is sampled independently from either the uniform distribution or the half-normal distribution, with marginals drawn from a Dirichlet distribution. Cost matrices drawn entrywise at random are standard in both studies regarding DOT problems [44] and related assignment problems [19]. In our experiments, we use this procedure to generate ten instances for each pair of marginals with sizes $N \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}\}$ with one of the two cost families, eighty in total. Cost matrices generated under this scheme have no underlying geometric structure, so no Point Cloud or Grid method applies to it, and it is only presented to demonstrate the efficacy of General format solvers.

Protocol. Every solve receives one A100-SXM4-80GB with its full 80 GB of device memory, 16 CPU cores with 93 GB of host memory, and a wall-clock limit of 1800 seconds; a solve that exceeds any of the three is recorded as such

and considered a failure. Every solver, including ours and baselines, runs to its *own native* stopping rule at its default settings. From each solver’s delivered state, we recover a representation of the primal–dual triple (X, α, β) for the original problem $(\mathcal{P})$ and score it by the KKT residual (2). Note that not every baseline implementation returns a pair of dual potentials (α, β) , so we derived mappings from each baseline’s returned state to their best estimate of the dual variables, i.e. solutions to $(\mathcal{D})$, in Appendix A. Furthermore, we measure two timings for each solve: t_{solve} stops when the solver returns at its native stopping criteria. t_{cert} adds to it any post-solve processing needed to obtain an optimality certificate in terms the KKT residual, which could involve the retrieval of the transport plan (HOT) or an expensive computation of the dual residual (HALO, DROT, M3S, and HOT). For solvers that terminate with a certificate of their own, we always have $t_{\text{solve}} = t_{\text{cert}}$. In Subsections 5.2 and 5.3, we present the comparison across solvers as an accuracy-time Pareto chart over the *co-solved* set of DOT instances to ensure that no solver is evaluated on a harder subset of problems than another. Each solver contributes a single pair $(t_{\text{cert}}, \text{KKT})$, with its t_{solve} marked by a thin line with a tick, when applicable. We occasionally score our solver under various values of input tolerances to map out a *Pareto frontier*.

DOT instances. DOT instances (C, r, c) across all three formats are presented in normalized form, where $\max(C) = 1$, and every solver of the same format receives the same data. Our solvers’ NORMALIZE steps are therefore always no-ops on benchmark input, but still performed in every solve to simulate a real use case. The marginal flooring procedure is internal to our three solvers and never applied to the data any baseline receives, or to the marginals any KKT residual is evaluated against.

Configuration of our solvers. All three of our solvers take two hyperparameters, the step size η and the check period T_{chk} . In our benchmarks, both are set by a fixed rule computed before the solve and held constant throughout it. The step size is determined by the rule $\eta = 0.7 \times \bar{C}$, where $\bar{C}$ denotes the mean entry of the cost matrix. The check period is a table on problem size, with $T_{\text{chk}} = 100, 200, 500, 1000, 2000$ and 4000 at $n_g = 32, 64, 128, 256, 512$ and 1024 , respectively. For every other instance, T_{chk} is read from the same table at the equivalent resolution $n_g = 2^{\lceil \log_2(N)/2 \rceil}$.

Baselines by format, and scaling limits. As discussed in Section 1, there is a wide spectrum of GPU-accelerated solvers for the DOT problem. Among them, we select eight to compare against our implementations of BDRS. Table 3 lists the eight solvers we have selected, as well as their respective DOT format, to be compared to the corresponding implementation of BDRS. Further, we provide the largest resolution at which each listed solver is able to consistently solve DOTmark problems at the stated protocol under its native stopping rule (for our solvers, $\text{tol} = 10^{-4}$). DOTmark resolution is the natural ruler for this because the dataset is Grid-native and lifts to all three formats, so that we

Table 3 The list of GPU-accelerated DOT solvers considered and their associated format, as well as the resolution of the largest DOTmark they can solve and why they cannot solve larger ones. The rows of our three solvers are for the stopping tolerance $\text{tol} = 10^{-4}$. *Full ladder*: every size was solved. *Timeout*: the next size hit the 1800 seconds limit on all or most instances attempted. *Device OOM* and *host OOM*: the next size exceeds the device or the host memory allocated to a solve. *Gate*: the next size was declined by the solver. *M3S requires its supports to be a grid but does not require a separable cost, so we classify it as a Point Cloud format solver for the purposes of our comparisons.

Solver	Format	Largest n_g solved	Why not larger
Grid BDRS (ours)	Grid	1024	full ladder
HOT	Grid	512	device OOM
Conv. Sinkhorn	Grid	1024	full ladder
Point Cloud BDRS (ours)	Point Cloud	256	timeout
HALO	Point Cloud	512	timeout
M3S	Point Cloud*	512	gate
General BDRS (ours)	General	256	device OOM
DROT	General	128	device OOM
PDOT	General	128	device OOM
cuPDLPx	General	64	host OOM
Sinkhorn	General	128	device OOM

can put every solver on the same scale. We concede that DOTmark steps are coarse ($16\times$ increase in number of decision variables per step), so the table gives rough estimates of scaling boundaries only.

5.2 The cost of generality across the three formats

A natural first question to ask would be: *does the computational cost of DOT across the three format differ significantly?* In this subsection, we answer this question affirmatively, and find that the result double as an overview of the time–accuracy performance of all the DOT solvers under consideration.

Figure 1 presents the Pareto plot of the solvers over the DOTmark instances at $n_g = 64$, with the ℓ_2^2 ground cost. We lift these instances to both Point Cloud and General, so that we can pass this set of instances to all the solvers listed in Table 3 in their desired format. Each of our three solvers were ran three times, at $\text{tol} = 10^{-4}, 10^{-6}, 10^{-8}$, respectively. We observe that at each tolerance, all three of our BDRS solvers terminate at the same accuracy that is under the given tolerance, which is a validation of the fact that they run identical iterations under mathematically equivalent representations. Moreover, Grid BDRS achieves a t_{cert} of 0.040 s, which is $11\times$ and $9\times$ faster than Point Cloud BDRS and General BDRS, respectively, demonstrating significant computational gaps even at the scale of $n_g = 64$.

Moreover, the Point Cloud solver consumes more structure than General and is nonetheless the slower of the two, by $1.19\times$ at this cell. The reason is the memory–recompute tradeoff between the two. The General solver needs to store a $m \times n$ cost matrix to read from it, while the Point Cloud solver recomputes it from coordinates every time it is needed so that the device need not be

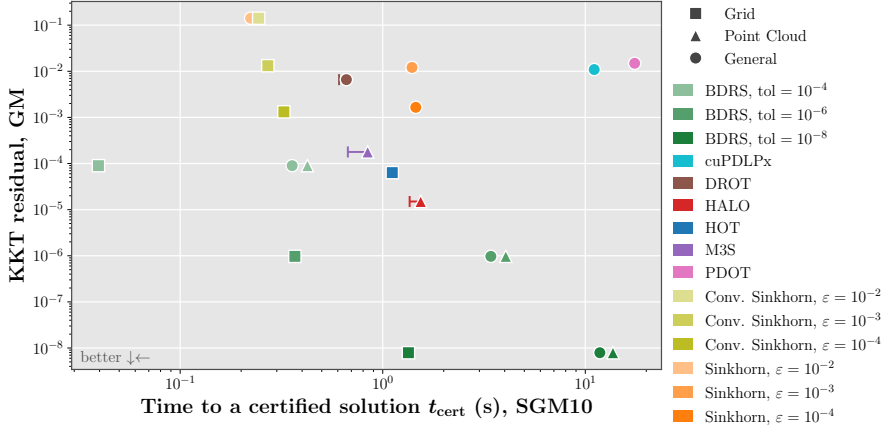


Fig. 1 Comparison of all solvers across all three formats on the same 414 co-solved DOT-mark instances at resolution $n_g = 64$ with ℓ_2^2 cost. We use a different color for each unique underlying algorithm, and use a different shape to indicate the format that the implementation consumes.

able to hold a $m \times n$ array at scale. For a practitioner the rule is therefore: use the Grid format when the cost is separable; otherwise materialize the cost and use the General solver whenever it fits, since it is faster wherever both run. Lastly, use the Point Cloud format when it does not fit, which on an 80 GB device begins near $N = 2^{17}$.

Zooming out, every Grid baseline solver is Pareto-dominated by our Grid solver at some choice of tolerance. Our General solver Pareto-dominates every General solver except Sinkhorn at $\varepsilon = 10^{-2}$, where our General solver at $\text{tol} = 10^{-4}$ achieves a geometric mean of KKT residual 1.6×10^3 times smaller than the Sinkhorn solution, at the cost of $1.6 \times$ runtime in SGM10 (0.36 against 0.22 s). Lastly, our Point Cloud solver at $\text{tol} = 10^{-4}$ Pareto-dominates M3S while achieving a $3.6 \times$ speedup over HALO at the cost of being $6.0 \times$ less accurate in terms of the geometric mean of KKT residual. In summary, our solvers provide a favorable accuracy–time tradeoff against the corresponding baseline solvers at every format.

5.3 Format-by-format comparison against the GPU baselines

Although Figure 1 provides a head-to-head comparison for every solver in Table 3, it does not test the scalability and the generalizability of the solvers. This subsection therefore performs in-depth comparisons between our implementations and the baselines on multiple datasets of varying sizes, in their respective format.

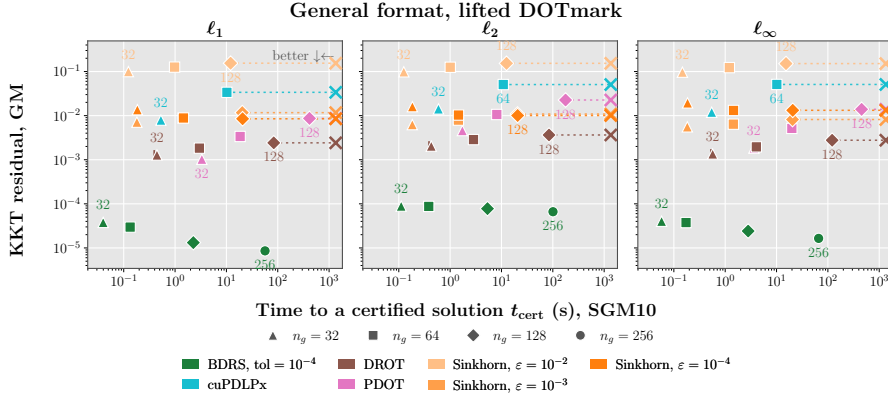


Fig. 2 Our BDRS General solver against the baseline General solvers on lifted DOTmark. In particular, the ℓ_1 , ℓ_2 , and ℓ_∞ norms were used at each size $n_g \in \{32, 64, 128, 256\}$. Every point aggregates all 450 instances of its choice of size and cost on DOTmark, except at $n_g = 128$ with the ℓ_1 and ℓ_∞ costs, where the co-solved sets hold the 447 and 442 instances on which PDOT finished inside the time limit. The cross on each panel marks cuPDLPx’s failed attempt at $n_g = 128$ due to a host-side OOM. Moreover, at $n_g = 256$, our General solver solve all instances across all norms to $\text{tol} = 10^{-4}$, while all baseline solvers fail the solve due to memory constraints.

5.3.1 The General format

Randomly generated cost and **lifted DOTmark** are used to benchmark the performance of solvers that take in General DOT instances (which we call “General solvers” for the remainder of this section).

Figure 2 compares the General solvers on **lifted DOTmark** at $n_g \in \{32, 64, 128, 256\}$, one panel per cost. Across the three, our solver is the fastest solver, among those that solve ($\mathcal{P}$) and those that solve its entropic approximation alike, at every size and cost, and the margin widens with problem size. The margin is the narrowest at $n_g = 32$ on the unsquared Euclidean cost, where DROT is $3.9\times$ slower (0.43 against 0.11 s) and the $\varepsilon = 10^{-2}$ Sinkhorn arm $1.1\times$ (0.13 s); at $n_g = 128$ ours leads DROT by $16\times$ to $43\times$ (84 to 121 s against 2.3 to 5.3 s), PDOT by $33\times$ to $183\times$ (177 to 454 s), and the three dense Sinkhorn arms by $2.3\times$ to $9.1\times$ (12 to 21 s). The widest speedup over DROT is on ℓ_∞ , where ours takes 2.8 s against 121 s; over PDOT and the $\varepsilon \in \{10^{-3}, 10^{-4}\}$ Sinkhorn arms, it is on ℓ_1 , where ours takes 2.3 s against 415 s and 20 to 21 s. Our General solver is also the most accurate solver in every cell, at a KKT of between 1.3 and 8.9×10^{-5} , against 1.3 to 3.6×10^{-3} for DROT, 1.0×10^{-3} to 2.3×10^{-2} for PDOT, and 5.6×10^{-3} to 1.5×10^{-1} for the three dense Sinkhorn arms. As a result, our General solver Pareto-dominates every other baseline solver, whether it solves ($\mathcal{P}$) or its entropic approximation, at every problem size, for every choice of cost on the DOTmark dataset.

At $n_g = 256$ our General solver continues to perform consistently while the baseline solvers face memory restrictions. In particular, our General solver

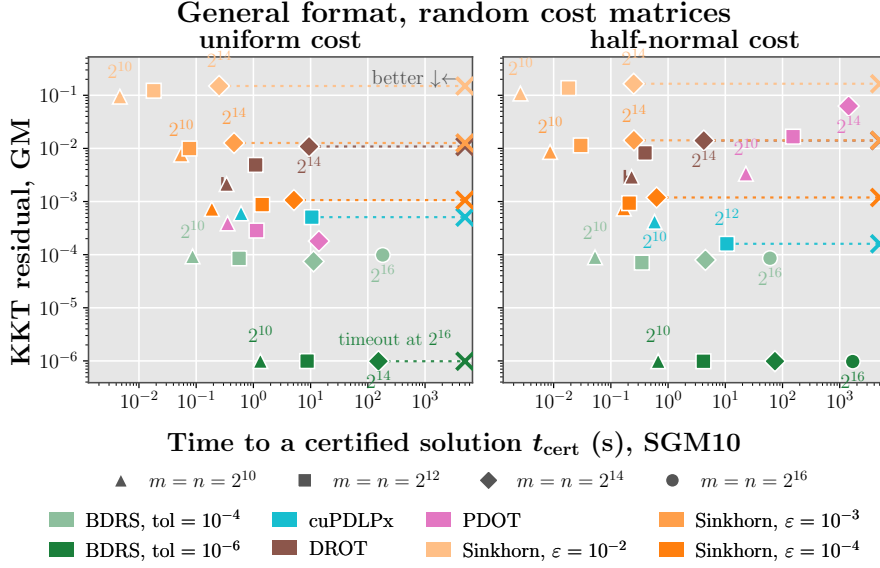


Fig. 3 Our General BDRS solver against the baseline General solvers on random cost matrices whose entries are drawn independently from the uniform distribution (left) and from the half-normal distribution (right), with ten instances per cost family at each size $m = n = N \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}\}$, with our solver at $\text{tol} \in \{10^{-4}, 10^{-6}\}$. On the half-normal cost, the points at $N = 2^{12}$ and 2^{14} aggregate only the 9 and 6 of the 10 instances on which PDOT finished inside the time limit. At $N = 2^{16}$, the half-normal points aggregate the 8 of the 10 instances on which our solver did not exceed device memory, and on the uniform cost, our $\text{tol} = 10^{-6}$ arm exceeds the time limit on all 10.

solves all 450 instances on each of the three costs, at a KKT between 8.6×10^{-6} and 6.6×10^{-5} , in 66, 57 and 101 s on ℓ_∞ , ℓ_1 and the unsquared Euclidean cost, respectively. This observation is consistent with the memory bottleneck recorded in Table 3.

Figure 3 presents the accuracy–time Pareto plots of the General solvers on **random cost matrices**. Our General solver Pareto-dominates the baseline General solvers of $(\mathcal{P})$ at 2^{10} and 2^{12} sizes for both uniform cost and half-normal cost. At $N = 2^{14}$ the picture changes for one baseline. cuPDLPx exceeds its host memory on both costs, and PDOT remains dominated on both panels. DROT, however, accepts a solution faster than our General solver, by 17 percent on the uniform cost (9.4 against 11.3 s) and by 6 percent on the half-normal cost (4.2 against 4.5 s), but at a KKT of 1.1 and 1.4×10^{-2} , more than two orders of magnitude above ours. Although neither solver dominates the other, we claim that our solver remains the favorable trade at that size. Moving on to the entropic solvers, $\varepsilon = 10^{-2}$ and $\varepsilon = 10^{-3}$ Sinkhorn implementations accepts a solution faster than ours everywhere, but our General solver overtakes the $\varepsilon = 10^{-4}$ Sinkhorn variant at $N = 2^{10}$ on both costs and at 2^{12} on the uniform cost. Moreover, as expected, no Sinkhorn variant produces a solution with a KKT less than 7.2×10^{-4} , and the quality of the

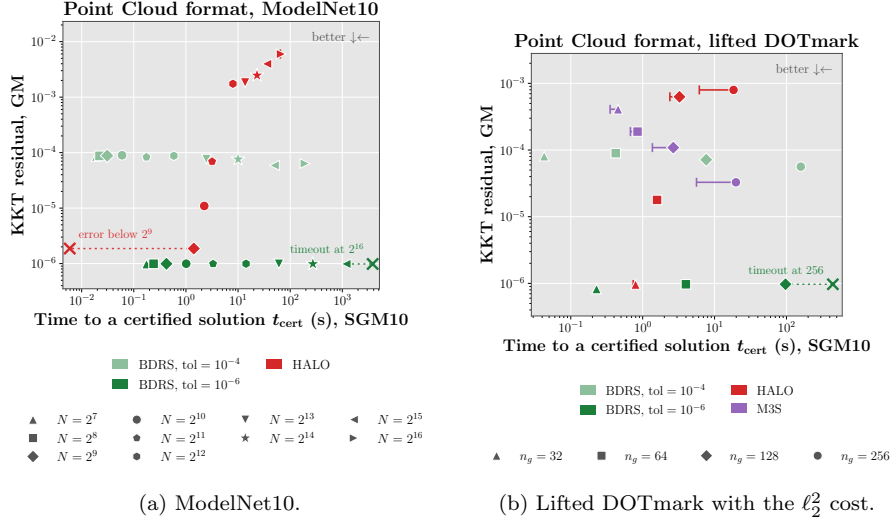


Fig. 4 Our Point Cloud solver with $\text{tol} \in \{10^{-4}, 10^{-6}\}$ against (a) HALO on ModelNet10 and (b) HALO and M3S on lifted DOTmark. (a) Our Point Cloud solver and HALO in its tree mode on ModelNet10 with the squared Euclidean cost, thirty pairs at each size from $N = 2^7$ to 2^{16} points per marginal. The cross at the left edge represents the fact that HALO rejects the 2^7 and 2^8 instances. (b) Our Point Cloud solver, HALO, and M3S on lifted DOTmark with the ℓ_2^2 cost, with resolutions $n_g \in \{32, 64, 128, 256\}$. The co-solved set holds 450 instances at $n_g = 32$ and 64, 439 at 128 and 370 at 256. The excluded instances are those on which M3S aborted the solve due to “too many absorptions” at $n_g = 128$ and $n_g = 256$. Our $\text{tol} = 10^{-6}$ arm reaches $N = 2^{15}$ in (a) and $n_g = 128$ in (b), the largest sizes at which it completes within the 1800 s budget. One of the thirty instances at $N = 2^{15}$ exceeds it and is excluded.

entropic solutions suffers approximately one order of magnitude for every order of magnitude ε increases. Meanwhile, our General solver achieves a KKT of less than 10^{-4} , as determined by tol , which is 7.8 to 2070 times lower than the Sinkhorn variants.

5.3.2 The Point Cloud format

ModelNet10 and **lifted DOTmark** are used to benchmark the performance of the Point Cloud solvers, namely our BDRS Point Cloud solver, HALO’s **tree** mode, and M3S. Notably, M3S further requires the support of the marginal distributions to lie on a grid, but does not require the cost to be separable. This puts M3S’ format specifications between the Grid and Point Cloud formats. For the purposes of this benchmark, we place it in the Point Cloud campaign. Figure 4a compares our Point Cloud solver against HALO’s **tree** mode on ModelNet10. HALO rejects instances with $N = 2^7$ and $N = 2^8$ and recommends the user to use an exact solver instead. Therefore, our solver is the only one that is able to solve ModelNet10 instances with sizes below $N = 2^9$, where we do so successfully up to $\text{tol} = 10^{-6}$ for all instances. For $N \in 2^9, \dots, 2^{14}$, we accept a solution faster, by $46\times$ at 2^9 narrowing to $2.3\times$ at 2^{14} , and be-

yond it HALO is faster, by $1.4\times$ at 2^{15} and $2.8\times$ at 2^{16} . However, our accepted solutions remain more accurate throughout. Concretely, our KKT residual consistently stays below the desired tolerance, at 10^{-4} and 10^{-6} , while HALO, being an approximation scheme, accepts solutions with KKT residual that increases monotonically with N , ranging from 1.9×10^{-6} to 6.0×10^{-3} at $N = 2^9$ to $N = 2^{16}$.

Figure 4b runs the same comparison on lifted DOTmark with M3S added, and the pattern repeats. At $n_g = 32$ and 64 , our $\text{tol} = 10^{-4}$ arm accepts a solution $19\times$ and $3.8\times$ faster than HALO and $11\times$ and $2.0\times$ faster than M3S; at $n_g = 128$ and 256 both baselines are faster than we are, HALO by $2.4\times$ and $8.6\times$ and M3S by $2.9\times$ and $8.0\times$. Our KKT residual is again flat across the four sizes, between 5.6×10^{-5} and 9.0×10^{-5} , while HALO's rises from 9.6×10^{-7} to 8.0×10^{-4} . M3S, which solves the entropic problem rather than $(\mathcal{P})$, is the least accurate of the three at the two smaller sizes, at 4.1×10^{-4} and 1.9×10^{-4} . At $n_g = 128$ and $n_g = 256$, M3S aborts the solve for 11 and 80 instances out of the 450 total instances, while HALO and our Point Cloud solver solve all 450. Among the co-solved problems at $n_g = 128$ and $n_g = 256$, M3S demonstrates comparable-to-better performance both in terms of accuracy and runtime. In summary, as the only exact solver in the Point Cloud format, our Point Cloud solver exhibits superior performance both in terms of runtime and accuracy at small to medium instances, and is the slower but more accurate choice at larger scales. Among the approximation solvers, HALO scales the best in runtime but rejects smaller instances and has degrading solution accuracy, while M3S achieves greater accuracy as problem size increases but becomes unreliable at medium-to-large scales.

5.3.3 The Grid format

DOTmark is used to benchmark the performance of solvers that take in Grid DOT instances ("Grid solvers"), at $n_g \in \{32, 64, 128, 256, 512, 1024\}$ with the squared Euclidean and the ℓ_1 cost. The Grid solvers consist of our Grid BDRS solver at $\text{tol} \in \{10^{-4}, 10^{-6}, 10^{-8}\}$, HOT, whose reduced flow model accepts the squared Euclidean cost only, and the convolutional Sinkhorn at $\varepsilon \in \{10^{-2}, 10^{-3}, 10^{-4}\}$.

Figure 5 compares the Grid solvers on **DOTmark** at $n_g \in \{32, \dots, 1024\}$, one panel per cost. Against HOT, our solver is Pareto-dominant at every size. From $n_g = 128$ to 512 the $\text{tol} = 10^{-4}$ arm is both faster (by $12\times$ to $14\times$) and more accurate (HOT's KKT rising from 2.3×10^{-4} to 2.8×10^{-3} while ours stay below 10^{-4}) than HOT. At $n_g = 32$ and $n_g = 64$, where HOT's KKT of 1.6×10^{-5} and 6.5×10^{-5} is below our native tolerance, the $\text{tol} = 10^{-6}$ variant of our Grid solver is both faster (0.12 and 0.36 s against 1.16 and 1.12 s) and 19 and 67 times more accurate. Lastly, at $n_g = 1024$, HOT runs out of device memory while ours finishes in 89 s for all 450 instances. At that size, only our $\text{tol} = 10^{-4}$ arm completes within the budget. Under the same 1800 s limit, the 10^{-6} arm reaches its tolerance on 252 of the 450 instances with the ℓ_1 cost and

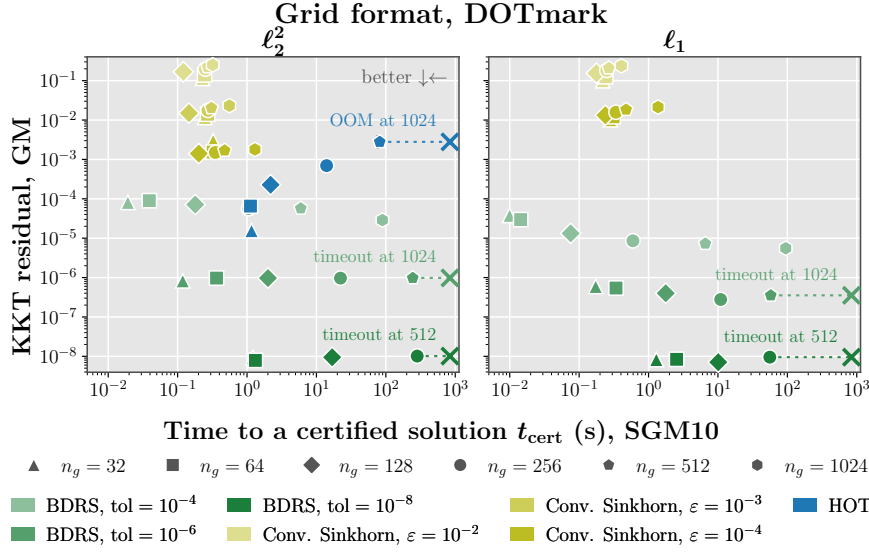


Fig. 5 The Grid solvers on DOTmark at $n_g = 32$ to 1024, with the ℓ_2^2 cost (left) and the ℓ_1 cost (right). HOT only accepts the ℓ_2^2 cost, and therefore appears only on the left panel. Our Grid solver is drawn at $\text{tol} \in \{10^{-4}, 10^{-6}, 10^{-8}\}$, and the convolutional Sinkhorn at $\varepsilon \in \{10^{-2}, 10^{-3}, 10^{-4}\}$. Each point aggregates all 450 co-solved instances of its size. The $\varepsilon = 10^{-4}$ Conv. Sinkhorn points with the ℓ_2^2 cost aggregate 285 to 366 instances per size, with the rest being solve failures due to the returned transport plan being non-finite. With the ℓ_1 cost it return non-finite values on more than half of every size and therefore is not drawn. The cross marks HOT at $n_g = 1024$, out of device memory on every instance. HOT’s certificate lines are hidden under its markers at this scale. The four remaining crosses mark our Grid solver at the first size each fails to complete within the 1800 s budget.

a minority of runs for the ℓ_2^2 cost. The 10^{-8} arm stops one size earlier still, at $n_g = 256$.

Convolutional Sinkhorn sacrifices accuracy and reliability for fast runtimes at large scales. In particular, its t_{cert} barely moves with size, every entropic point lying between 0.12 and 1.4 s while N grows 1024-fold, so it overtakes our $\text{tol} = 10^{-4}$ arm from $n_g = 128$ upward with the ℓ_2^2 cost and from $n_g = 256$ upward with ℓ_1 . However, no entropic point with either cost fall below $\text{KKT} = 1.4 \times 10^{-3}$, and at small values of ε , it stops being reliable. Concretely, the $\varepsilon = 10^{-2}$ and 10^{-3} variants solve all 450 instances with both costs, but the KKT residuals their solutions achieve sit between 1.0×10^{-2} and 2.5×10^{-1} , at least two orders of magnitude above ours. Its most accurate arm, the $\varepsilon = 10^{-4}$ variant, returns a non-finite plan on 19 to 37 percent of the instances with the squared Euclidean cost and on 65 to 90 percent with ℓ_1 , the latter of which is therefore not drawn on the right panel. By contrast, our Grid solver returned a certified solution on all 450 instances of every point drawn. Moreover, at $n_g \leq 64$ on both costs and at $n_g = 128$ for ℓ_1 , our Grid solver at $\text{tol} = 10^{-4}$ Pareto-dominates all convolutional Sinkhorn variants. Notably, at $n_g = 32$ with the ℓ_1 cost, even our solver running at $\text{tol} = 10^{-6}$ runs faster than every

entropic arm (0.17 s against 0.22 s and up), and at $n_g = 64$ it is within $1.4\times$ of the fastest of them (0.34 s against 0.24 s). At both sizes its KKT is more than three orders of magnitude below any entropic point. In summary, our Grid solver at $\text{tol} = 10^{-4}$ is the only solver that solves all 450 DOTmark instances at every size with both costs. It Pareto-dominates HOT at every size on the ℓ_2^2 cost, the only cost HOT accepts, and it dominates every convolutional Sinkhorn arm up to $n_g = 64$ on both costs and up to $n_g = 128$ on the ℓ_1 cost. Past those sizes, convolutional Sinkhorn’s runtime scales better with problem size, but none of them reaches a KKT residual below 1.4×10^{-3} , and the one that comes closest, $\varepsilon = 10^{-4}$, returns a non-finite plan on up to 37% of the ℓ_2^2 instances and on more than half of the ℓ_1 ones.

5.4 Ablation and robustness studies

In this subsection, we demonstrate the dual stall phenomenon of vanilla BDRS and perform ablation studies on the c-transform post-processing on the various Douglas–Rachford splitting-based solvers to demonstrate its effectiveness on BDRS. Then, we examine the robustness of BDRS with respect to the step size parameter η and the termination check period T_{chk} .

5.4.1 Dual stall phenomenon and c-transform ablation.

Section 3.2 motivates the c-transform post-processing by the observation that r_D lags behind the other two residuals. Figure 6 quantifies that lag on a subset of DOTmark for the three Douglas–Rachford solvers (BDRS, DROT, HOT). For BDRS and DROT, the KKT residual (2) of the native iterates, which is defined as the maximum of its three residuals, is equal to r_D^k at more than 98% of the checkpoints. This is the aforementioned “dual stall”. At the check where the corrected triple first meets the tolerance, the dual residual is 8.2×10^3 times (resp. 5.3×10^4 times) above the larger of the other two terms for BDRS at $n_g = 32$ (resp. 64), and 21 times (resp. 50 times) for DROT. The dual stall is sharper for BDRS, as the primal residual and the native gap are 236 and 540 times below the tolerance, while the dual is 35 times above it. As a result, the green curves (representing r_D^k) on panels (a), (b), (d), and (e) can also be interpreted as the native KKT residual, which always lags behind the corrected KKT residual, in magenta, demonstrating that the c-transform post-processing shortens the solve. More specifically, it shortens it by a median factor of 7.5 (resp. 9.1) for BDRS at $n_g = 32$ (resp. 64). For DROT the native triple never reaches the tolerance, so we can only lower bound the speedup factor by 1.4 (resp. 1.8), which is computed as the length of the iteration window divided by the corrected crossing. For HOT, the largest term starts as r_D^k but changes to $|g^k|$ at a median $k = 1000$ and 1600 at the two sizes, at which point the native KKT residual and the corrected KKT residual becomes equal. Before that crossover, HOT’s dual stall is of the same order as ours, 707 times (resp. 2827 times) the larger of the other two terms at $n_g = 32$

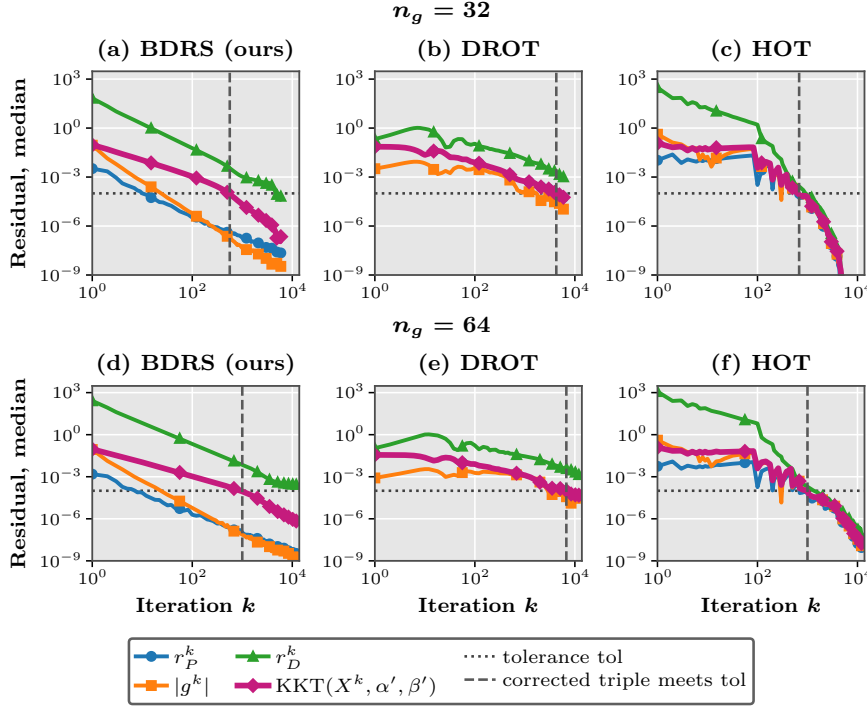


Fig. 6 The median of various residuals against iteration count for the three Douglas–Rachford solvers (BDRS, DROT, HOT), on the same 50 DOTmark instances, with the ℓ_2^2 cost and $\text{tol} = 10^{-4}$, at $n_g = 32$ in panels (a), (b), and (c) (resp. $n_g = 64$ in panels (d), (e), and (f)). Each solver run for a fixed 6000 (resp. 12000) iterations with no stopping rule and scored at 101 (resp. 163) checks from iteration 1. On all six panels, the blue, green, and orange curves are the medians of r_P^k , r_D^k , and $|g^k|$ for the triple (X^k, α^k, β^k) , respectively. The magenta curve is the median KKT residual of the corrected triple (X^k, α', β') , whose duals are the c-transformed *corrected duals*. Note that the corrected duals are never fed back into the iterations and only serves as a termination check proxy.

(resp. $n_g = 64$). What separates HOT is that its stall then decays, to 182 and 712 times at $k = 100$ and to 2.4 and 21 times at $k = 500$, while ours grows to 9.2×10^3 and 3.5×10^4 at $k = 100$. As a result, the c-transform post-processing has a limited effect on HOT, shortening the solve by a median factor of 1.4 and 1.5 rather than the 7.5 and 9.1 for BDRS.

5.4.2 Robustness of BDRS with respect to hyperparameters.

There are two hyperparameters for our solvers: the step-size η and the check period T_{chk} . We provide a provisional default selection scheme for both, and leave the precise tuning of these parameters as future work. The deployed rules for both are described in Section 5.1.

Figure 7 shows how the iteration count and solve time of our algorithm move as we perturb each parameter on a subset of DOTmark instances. We are

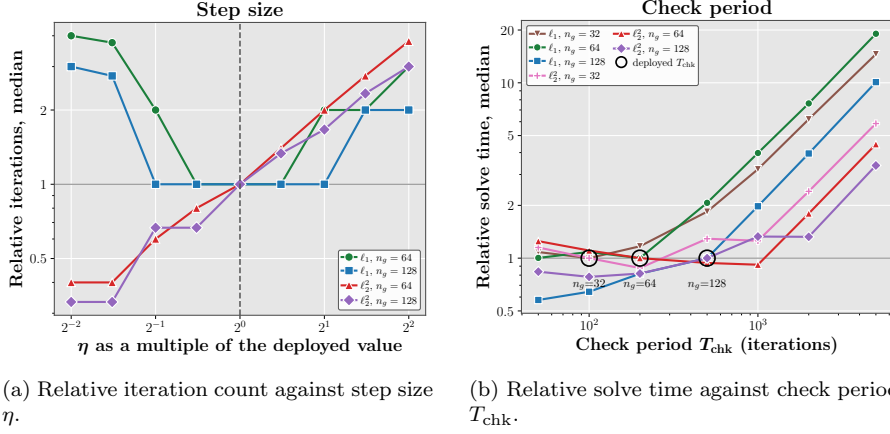


Fig. 7 Sensitivity of our solvers to the (a) step size η and (b) check period T_{chk} , on DOT-mark at $\text{tol} = 10^{-4}$. Each solve's iteration count in (a), and its solve wall time in (b), is divided by the same quantity of the same instance at the deployed value of the swept constant, and the solid horizontal line marks a ratio of one. For both panels, one curve is plotted per choice of ground cost and resolution. (a) resolutions $n_g \in \{64, 128\}$ was used, with 50 instances each (five sampled pairs from all ten classes). The dashed vertical line marks the production value. (b) The production η was used at T_{chk} ranging from 50 to 5000. Resolutions $n_g \in \{32, 64, 128\}$ was used, with 30 instances each (three sampled pairs from all ten classes). The rings mark the deployed period of each resolution, 100, 200 and 500 at $n_g = 32, 64$ and 128, where each curve is one by construction.

interested in how much the performance of our algorithm changes when we perturb each parameter. In Figure 7a, we sweep the value of η across a factor of sixteen, from a quarter of the deployed value to four times it. We find that our solvers are able to solve all instances in every perturbation, while the median iteration count stays inside $[0.33, 4.0]$ times the count that the solve took to solve the instances at the deployed value. Moreover, it is clear that the deployed η is not the optimal choice for the ℓ_2^2 instances of this subset, since we observe a linear trend between η and the median of the iteration count required to solve them, but we emphasize that despite this suboptimality we still achieve the scalability demonstrated in Section 5.3.

Next, in Figure 7b, we sweep T_{chk} across a hundred-fold span, from 50 to 5000 iterations. Every perturbation solves every instance. The median solve time is flat from $T_{\text{chk}} = 50$ to $T_{\text{chk}} = 500$. At $T_{\text{chk}} \geq 1000$, which represents an overly loose check period at the tested problem sizes, the solver mostly terminates at the first termination check, causing wasted iterations. On the contrary, for $T_{\text{chk}} \leq 50$, a period that is slightly shorter than preferred for the tested problem sizes, we indeed terminate at fewer total iterations, but at the cost of more frequent checks. More specifically, at $T_{\text{chk}} = 50$, the median solve takes about 16 percent fewer iterations than at the deployed period and yet 5 percent more time.

6 Conclusion

In this paper, we presented three GPU implementations of the BDRS algorithm for solving the DOT problems with three data formats. We proposed a c-transform post-processing step that produces exactly feasible dual variables. This procedure helped accelerate the convergence of BDRS. We conducted extensive numerical experiments to compare our GPU-accelerated BDRS with existing competitive solvers and the results showed that our implementations demonstrated superior performance. We leave it as a future work to distribute the solve over multiple GPU devices as suggested by the recent developments in multi-GPU mathematical programming solvers [27, 20].

References

1. Alaya, M.Z., Bérar, M., Gasso, G., Rakotomamonjy, A.: Screening Sinkhorn algorithm for regularized optimal transport. In: *Advances in Neural Information Processing Systems*, vol. 32 (2019)
2. Altschuler, J., Weed, J., Rigollet, P.: Near-linear time approximation algorithms for optimal transport via Sinkhorn iteration. In: *Advances in Neural Information Processing Systems*, vol. 30, pp. 1964–1974 (2017)
3. Applegate, D., Díaz, M., Hinder, O., Lu, H., Lubin, M., O’Donoghue, B., Schudy, W.: Practical large-scale linear programming using primal-dual hybrid gradient. In: *Advances in Neural Information Processing Systems*, vol. 34, pp. 20243–20257 (2021)
4. Auricchio, G., Bassetti, F., Gualandi, S., Veneroni, M.: Computing Kantorovich–Wasserstein distances on d-dimensional histograms using (d+1)-partite graphs. In: *Advances in Neural Information Processing Systems*, vol. 31 (2018)
5. Baretto, R., Della Croce, F., Scatamacchia, R.: An improved variant of the iterated inside out algorithm for solving the optimal transport DOTmark instances. *Mathematical Programming Computation* (2026). DOI 10.1007/s12532-026-00341-2
6. Baretto, R., Della Croce, F., Scatamacchia, R.: Iterated inside out: a new exact algorithm for the transportation problem. *INFORMS Journal on Computing* **38**(1), 39–52 (2026). DOI 10.1287/ijoc.2024.0642
7. Bertsekas, D.P.: *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, New York (1982)
8. Bertsekas, D.P.: The auction algorithm: a distributed relaxation method for the assignment problem. *Annals of Operations Research* **14**(1), 105–123 (1988). DOI 10.1007/BF02186476
9. Blanchard, P., Higham, D.J., Higham, N.J.: Accurately computing the log-sum-exp and softmax functions. *IMA Journal of Numerical Analysis* **41**(4), 2311–2330 (2021). DOI 10.1093/imanum/draa038
10. Bonneel, N., van de Panne, M., Paris, S., Heidrich, W.: Displacement interpolation using Lagrangian mass transport. *ACM Transactions on Graphics* **30**(6), 158:1–158:12 (2011). DOI 10.1145/2070781.2024192
11. Bregman, L.M.: The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. *USSR Computational Mathematics and Mathematical Physics* **7**(3), 200–217 (1967)
12. Censor, Y., Zenios, S.A.: Proximal minimization algorithm with *D*-functions. *Journal of Optimization Theory and Applications* **73**(3), 451–464 (1992)
13. Chen, Y., Li, C., Lu, Z.: Computing wasserstein-p distance between images with linear cost. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 519–528 (2022)
14. Choquette, J.: NVIDIA Hopper H100 GPU: Scaling performance. *IEEE Micro* **43**(3), 9–17 (2023). DOI 10.1109/MM.2023.3256796

15. Choquette, J., Gandhi, W., Giroux, O., Stam, N., Krashinsky, R.: NVIDIA A100 tensor core GPU: Performance and innovation. *IEEE Micro* **41**(2), 29–35 (2021). DOI 10.1109/MM.2021.3061394
16. Courty, N., Flamary, R., Tuia, D., Rakotomamonjy, A.: Optimal transport for domain adaptation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **39**(9), 1853–1865 (2017). DOI 10.1109/TPAMI.2016.2615921
17. Cuturi, M.: Sinkhorn distances: Lightspeed computation of optimal transport. In: *Advances in Neural Information Processing Systems*, vol. 26 (2013)
18. Cuturi, M., Meng-Papaxanthos, L., Tian, Y., Bunne, C., Davis, G., Teboul, O.: Optimal transport tools (OTT): a JAX toolbox for all things Wasserstein. *arXiv preprint arXiv:2201.12324* (2022)
19. Dell’Amico, M., Toth, P.: Algorithms and codes for dense assignment problems: the state of the art. *Discrete Applied Mathematics* **100**(1-2), 17–48 (2000). DOI 10.1016/S0166-218X(99)00172-9
20. Deza, A., Dey, S., Van Hentenryck, P.: Distributed linear programming on GPU clusters at extreme scale. *arXiv preprint arXiv:2609.09108* (2026)
21. Douglas Jr., J., Rachford Jr., H.H.: On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society* **82**(2), 421–439 (1956)
22. Eckstein, J.: Nonlinear proximal point algorithms using Bregman functions, with applications to convex programming. *Mathematics of Operations Research* **18**(1), 202–226 (1993)
23. Eckstein, J., Bertsekas, D.P.: On the Douglas–Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming* **55**(1–3), 293–318 (1992)
24. Flamary, R., Courty, N., Gramfort, A., et al.: POT: Python optimal transport. *Journal of Machine Learning Research* **22**(78), 1–8 (2021)
25. IEEE: IEEE standard for floating-point arithmetic. *IEEE Std 754-2019* (Revision of IEEE Std 754-2008) (2019). DOI 10.1109/IEEESTD.2019.8766229
26. Kuhn, H.W.: The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* **2**(1–2), 83–97 (1955). DOI 10.1002/nav.3800020109
27. Li, H., Huang, Y., Liu, H., Ge, D., Ye, Y.: D-pdip: Scaling pdip to distributed multi-gpu systems. *arXiv preprint arXiv:2601.07628* (2026)
28. Ling, H., Okada, K.: An efficient earth mover’s distance algorithm for robust histogram comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**(5), 840–853 (2007). DOI 10.1109/TPAMI.2007.1058
29. Lions, P.L., Mercier, B.: Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis* **16**(6), 964–979 (1979)
30. Lu, H., Peng, Z., Yang, J.: cupdlpx: A further enhanced gpu-based first-order solver for linear programming. *arXiv preprint arXiv:2507.14051* (2025)
31. Lu, H., Yang, J.: Pdot: A practical primal-dual algorithm and a gpu-based solver for optimal transport. *arXiv preprint arXiv:2407.19689* (2024)
32. Ma, S.: Convergence of BDRS as a matrix scaling algorithm. *Optimization Online* (2026)
33. Ma, S., Xiao, L., Zhao, R.: Bregman douglas-rachford splitting method. *arXiv preprint arXiv:2509.08739* (2025)
34. Mai, V.V., Lindbäck, J., Johansson, M.: A fast and accurate splitting method for optimal transport: Analysis and implementation. *arXiv preprint arXiv:2110.11738* (2021)
35. Milakov, M., Gimelshein, N.: Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867* (2018)
36. Munkres, J.: Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics* **5**(1), 32–38 (1957). DOI 10.1137/0105003
37. NVIDIA Corporation: NVIDIA A100 tensor core GPU datasheet. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf> (2021). Accessed 2026-09-21
38. Orlin, J.B.: A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming* **78**(2), 109–129 (1997). DOI 10.1007/BF02614365

39. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: *Advances in Neural Information Processing Systems* 32 (2019)
40. Peyré, G., Cuturi, M.: Computational optimal transport: with applications to data science. *Foundations and Trends in Machine Learning* **11**(5–6), 355–607 (2019). DOI 10.1561/22000000073
41. Rubner, Y., Tomasi, C., Guibas, L.J.: The earth mover’s distance as a metric for image retrieval. *International Journal of Computer Vision* **40**(2), 99–121 (2000). DOI 10.1023/A:1026543900054
42. Schmitzer, B.: Stabilized sparse scaling algorithms for entropy regularized transport problems. *SIAM Journal on Scientific Computing* **41**(3), A1443–A1481 (2019). DOI 10.1137/16M1106018
43. Schrieber, J., Schuhmacher, D., Gottschlich, C.: Dotmark—a benchmark for discrete optimal transport. *IEEE Access* **5**, 271–282 (2016)
44. Schwinn, J., Werner, R.: On the effectiveness of primal and dual heuristics for the transportation problem. *IMA Journal of Management Mathematics* **30**(3), 281–303 (2019). DOI 10.1093/imaman/dpy011
45. Solomon, J., de Goes, F., Peyré, G., Cuturi, M., Butscher, A., Nguyen, A., Du, T., Guibas, L.: Convolutional Wasserstein distances: efficient optimal transportation on geometric domains. *ACM Transactions on Graphics* **34**(4), 66:1–66:11 (2015). DOI 10.1145/2766963
46. Tseng, P., Bertsekas, D.P.: On the convergence of the exponential multiplier method for convex programming. *Mathematical Programming* **60**(1), 1–19 (1993)
47. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* **52**(4), 65–76 (2009). DOI 10.1145/1498765.1498785
48. Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., Xiao, J.: 3D ShapeNets: a deep representation for volumetric shapes. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015)
49. Xia, W., Zhu, Y.N., Liang, J., Zhang, X.: A memory-efficient hierarchical algorithm for large-scale optimal transport problems. In: *The Fourteenth International Conference on Learning Representations* (2026)
50. Zhang, G., Gu, Z., Yuan, Y., Sun, D.: Hot: An efficient halpern accelerating algorithm for optimal transport problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2025)

A Recovering primal–dual pairs from the baseline solvers

The cross-solver comparison of Section 5 scores every solver’s returned solution by the KKT residual (2) on its *corresponding* primal–dual triple (X, α, β) for the *C-normalized* version of $(\mathcal{P})$. Each baseline solver terminates based on its own stopping rule, and for some baseline solvers, the mapping from their stopping rule to the KKT residual is not clear. The plan X is in every case the one the solver itself returns, so this appendix records, for each baseline, only how the dual pair (α, β) is obtained from the solver’s final state. For entropic solvers, we follow the convention of [31] and score the entropic potentials as the dual potentials of $(\mathcal{P})$ directly.

DROT. DROT [34] applies Euclidean Douglas–Rachford splitting to $(\mathcal{P})$. Alongside the plan it maintains two vectors $\phi_1^k \in \mathbb{R}^m$ and $\phi_2^k \in \mathbb{R}^n$, which enter its iterations [34, Algorithm 1] as

$$X^{k+1} = (X^k + \phi_1^k \oplus \phi_2^k - \sigma C)_+$$

with step-size $\sigma > 0$. [34] show that this pair implicitly plays the role of the dual variables of $(\mathcal{D})$: $(\phi_1^k/\sigma) \oplus (\phi_2^k/\sigma)$ converges to $\alpha^* \oplus \beta^*$, and the method’s own stopping criteria are the primal residual, the dual residual and the duality gap built from $(\phi_1^k/\sigma, \phi_2^k/\sigma)$. Our recovery reads the two vectors at DROT’s native stop and rescales,

$$(\alpha, \beta) = (\phi_1/\sigma, \phi_2/\sigma),$$

with $\sigma = 2/(m + n)$ the implementation’s step-size.

HOT. HOT [50] does not solve $(\mathcal{P})$ directly. On an $h \times w$ grid of pixels with a squared Euclidean ground cost, it solves the reduced flow LP of [4], in which every unit of mass moves in two parts: first within its row, then within its column. Denoting a pixel’s coordinates as (p, q) and the separable cost as $C_{ab} = \varphi_c(|q - q'|) + \varphi_r(|p - p'|)$ for $a = (p, q)$, $b = (p', q')$, the reduced LP has nonnegative variables $x_{p;q \rightarrow q'}^1$ (a horizontal move within row p , cost $\varphi_c(q, q')$) and $x_{q';p \rightarrow p'}^2$ (a vertical move within column q' , cost $\varphi_r(p, p')$), and three sets of constraints:

$$\sum_{q'} x_{p;q \rightarrow q'}^1 = r_{(p,q)} \quad (\text{consistency at source, one per pixel } (p, q)), \quad (13)$$

$$\sum_q x_{p;q \rightarrow q'}^1 = \sum_{p'} x_{q';p \rightarrow p'}^2 \quad (\text{incoming} = \text{outgoing mass, one per node } (p, q')), \quad (14)$$

$$\sum_p x_{q';p \rightarrow p'}^2 = c_{(p',q')} \quad (\text{consistency at target, one per pixel } (p', q')). \quad (15)$$

This reduces the variable count from $h^2 w^2$ to $hw(h + w)$. Let $(\lambda, \bar{\alpha}, \bar{\beta})$ denote the duals of (14), (13), and (15) respectively. The dual constraints of the reduced LP read

$$\bar{\alpha}_{(p,q)} + \lambda_{(p,q')} \leq \varphi_c(|q - q'|), \quad \bar{\beta}_{(p',q')} - \lambda_{(p,q')} \leq \varphi_r(|p - p'|). \quad (16)$$

Proposition 2. *If $(\lambda, \bar{\alpha}, \bar{\beta})$ is feasible for the reduced dual, then $(\alpha, \beta) := (\bar{\alpha}, \bar{\beta})$ is feasible for $(\mathcal{D})$, with the same dual objective value. Further, an optimal reduced dual $(\lambda^*, \bar{\alpha}^*, \bar{\beta}^*)$ maps to an optimal (α^*, β^*) for $(\mathcal{D})$.*

Proof. Fix a source $a = (p, q)$ and a target $b = (p', q')$. Since we always transport mass horizontally then vertically, every route from a to b must pass through the node (p, q') . Adding the two constraints of (16) along this route,

$$\bar{\alpha}_a + \bar{\beta}_b \leq \varphi_c(|q - q'|) + \varphi_r(|p - p'|) = C_{ab}.$$

Every constraint of the original dual is thus the sum of exactly two reduced constraints. As a result, feasible reduced duals always yield feasible dual variables of $(\mathcal{D})$. Since the coupling

block (14) has zero right-hand side, the reduced dual objective equals $r^\top \bar{\alpha} + c^\top \bar{\beta}$, which is the objective of $(\mathcal{D})$ at (α, β) . Lastly, since the reduced LP and $(\mathcal{P})$ have equal optimal values [4, Thm. 1 and Cor. 1], strong duality on both the reduced LP and $(\mathcal{D})$ makes (α, β) optimal for $(\mathcal{D})$ whenever the reduced dual is optimal for the reduced LP. $\square$

Writing c_{LP} and b for the reduced LP's cost vector and right-hand side, HOT's preconditioning procedure internally rescales them by $1/c_s$ and $1/b_s$, respectively, with $c_s = \|c_{\text{LP}}\|_2 + 1$ and $b_s = \|b\|_2 + 1$. A right-hand-side rescaling leaves the dual solution unchanged while a cost rescaling scales it linearly, so the recovery mapping multiplies HOT's returned dual vector by c_s alone. Composing this with Proposition 2 gives the mapping used in our code, $(\lambda, \bar{\alpha}, \bar{\beta}) \mapsto (c_s \bar{\alpha}, c_s \bar{\beta})$.

HALO. HALO [49] is a coarse-to-fine restricted-support method. Both marginal distributions r and c are coarsened through a procedure that merges pairs of supports next to each other. At each level of coarseness, the method solves the LP restricted to a sparse candidate active set while retaining all $m + n$ marginal equality constraints [49, Def. 1]. cuPDLPx [30] is used as the inner LP solver. Then, HALO prices the excluded entries with the resulting dual pair $(u, v) \in \mathbb{R}^m \times \mathbb{R}^n$, appending those of most negative reduced cost $u_i + v_j - C_{ij}$ and re-solving. Because the constraint set is never modified, every inner LP's dual has full length $m + n$, so a dual for the original problem is available at the end of the inner solve for any level. Therefore, at HALO's native stop, our recovery map should read the finest-level dual variables, reversing any additional scaling that HALO performs internally. In fact, HALO applies scaling only in its Grid mode, where $C_{ij}^{\text{HALO, Grid}} = ((p - p')^2 + (q - q')^2)/n_g^2$. There is no scaling in its Point Cloud (Tree) mode. Therefore, our recovery map from HALO's Grid mode is $(u^{\text{Grid}}, v^{\text{Grid}}) \mapsto (n_g^2 u^{\text{Grid}}/(2(n_g - 1)^2), n_g^2 v^{\text{Grid}}/(2(n_g - 1)^2))$, while the map for Point Cloud (Tree) mode is $(u^{\text{Tree}}, v^{\text{Tree}}) \mapsto (u^{\text{Tree}}, v^{\text{Tree}})$.

PDOT and cuPDLPx. Both PDOT [31] and cuPDLPx [30] return the dual of the original problem that's passed to them. Since we normalize C before passing it to our solvers, we can take the dual variables returned by both PDOT and cuPDLPx as-is.

Log-domain Sinkhorn, Grid Sinkhorn, and M3S. Log-domain Sinkhorn [42], as implemented in POT [24], maintains the log-scalings (u, v) of the plan $X = \exp(-C/\varepsilon + u \oplus v)$, where ε is the regularization parameter. It iterates as follows

$$u_i \leftarrow \log r_i - \text{LSE}_j(-C_{ij}/\varepsilon + v_j), \quad v_j \leftarrow \log c_j - \text{LSE}_i(-C_{ij}/\varepsilon + u_i). \quad (17)$$

Setting $(\alpha, \beta) := (\varepsilon u, \varepsilon v)$ gives

$$\alpha_i + \beta_j - C_{ij} = \varepsilon \log X_{ij}. \quad (18)$$

Let us denote $H(X) := -\sum_{ij} X_{ij} \log X_{ij}$. Throughout this paragraph, r_P , r_D and g are the primal residual, the dual residual and the duality gap of the *original* problem $(\mathcal{P})$. Firstly, after any row (resp. column) update of (17), the corresponding marginal sum agrees exactly with r (resp. c). Since the entries are nonnegative and r and c are probability vectors, each entry is bounded by its own marginal, $X_{ij} \leq r_i \leq 1$ (resp. $X_{ij} \leq c_j \leq 1$). Hence $\log X_{ij} \leq 0$, and (18) gives $\alpha \oplus \beta \leq C$ entrywise. This means that the recovered dual variables (α, β) are always feasible for $(\mathcal{D})$. Secondly, at a primal-feasible iterate, substituting $r = X \mathbf{1}_n$ and $c = X^\top \mathbf{1}_m$, (18) gives

$$r^\top \alpha + c^\top \beta = \sum_{ij} X_{ij} (\alpha_i + \beta_j) = \langle C, X \rangle + \varepsilon \sum_{ij} X_{ij} \log X_{ij},$$

so that the duality gap is

$$g = \langle C, X \rangle - r^\top \alpha - c^\top \beta = -\varepsilon \sum_{ij} X_{ij} \log X_{ij} = \varepsilon H(X).$$

This non-vanishing quantity $\varepsilon H(X)$ is the consequence of the Sinkhorn algorithm solving a regularized version of $(\mathcal{P})$ as opposed to the original.

Grid Sinkhorn [45], as implemented in OTT [18], runs the same iteration, so (18) and both consequences apply verbatim.

M3S [13] is a multiscale ε -scaling Sinkhorn. Unlike the two Sinkhorn solvers above, it carries the marginals in the exponential,

$$X_{ij} = r_i c_j \exp((\hat{\alpha}_i + \hat{\beta}_j - C_{ij})/\varepsilon), \quad (19)$$

with $(\hat{\alpha}, \hat{\beta})$ being its finest-level potentials. Taking logs in (19) and multiplying both sides by ε ,

$$\varepsilon \log X_{ij} = (\hat{\alpha}_i + \varepsilon \log r_i) + (\hat{\beta}_j + \varepsilon \log c_j) - C_{ij},$$

so the recovery map $(\hat{\alpha}, \hat{\beta}) \mapsto (\hat{\alpha} + \varepsilon \log r, \hat{\beta} + \varepsilon \log c)$ produces a pair satisfying (18) exactly, and the non-vanishing $\varepsilon H(X)$ still applies. Moreover, both potentials are then rescaled from M3S's native squared-pixel cost units by $1/(2(n_g - 1)^2)$, the same factor as its objective. Therefore, the dual recovery mapping we implement is

$$(\hat{\alpha}, \hat{\beta}) \mapsto \left(\frac{\hat{\alpha} + \varepsilon \log r}{2(n_g - 1)^2}, \frac{\hat{\beta} + \varepsilon \log c}{2(n_g - 1)^2} \right).$$

Lastly, M3S truncates its plan. More specifically, outside of a sparse support set, the plan is set to zero by construction rather than driven to zero asymptotically, so the identity constrains (α, β) only on that support and says nothing about the remaining entries. Dual feasibility there is therefore not automatic, and our experiments measure and report r_D as part of the KKT residual (2) on the delivered pair like any other.