# A Omitted Proofs For Matching Algorithm

## A.1 Proof of Invariants (I1) and (I2)

Note that (I1) and (I2) are true at the start of the algorithm. Inductively assume that the invariants hold at the start of any phase. We show that the invariants continue to hold at the end of the phase.

**Proof of (I1):** The dual update step (III) only increases the dual weight of $B$ and reduces the dual weight of $A$. Therefore, the dual weights of $A$ remain non-positive and the dual weights of $B$ remain non-negative. Next, we show that every free vertex of $A$ with respect to the matching $M$ continues to have a dual weight of 0 at the end of this phase. Clearly, step (I) does not affect this property. Step (II) updates the matching $M$. From Lemma 2.1, every matched vertex of $A$ remains matched after the execution of Step (II). So, every free vertex of $A$ continues to have a dual weight of 0. During step (III (a)), the dual weight for any vertex $a \in A$ reduces only if it is matched in $M'$. By construction, step (II) will add the edges of $M'$ to $M$. So, every vertex of $A$ whose dual weight is updated by (III) is matched in $M$ after updating $M$. Consequently, the dual weights of all free vertices of $A$ remain unchanged, i.e., their dual weight remains 0 implying (I1).

**Proof of (I2):** To show that the edge remains feasible, we need to show that the slack on every edge remains non-negative and all edges of $M$ have zero slack. During each phase, the feasibility conditions could potentially be affected for two reasons. First, the matching $M$ gets updated in step (II). Second, the dual weights of certain points change in Step (III). At the start of the phase, consider an edge $(a, b)$ that is feasible but not admissible, i.e., it has a slack of at least $\varepsilon$. The matching $M'$ consists only of admissible edges, and, therefore, $(a, b) \notin M$. Observe that the dual weight of $b$ may increase by $\varepsilon$ in Step (III(b)) reducing its slack by $\varepsilon$. The slack, however, remains non-negative. In Step (III(a)), the dual weight of $a$ may reduce by $\varepsilon$, only increasing the slack on $(a, b)$. Therefore, the edge $(a, b)$ continues to be feasible and satisfies (3).

Next, consider an edge $(a, b)$ that is admissible at the start of the phase. At the end of the phase, there are two possibilities: either $(a, b)$ is in the matching, or it is not in the matching. We begin by showing that if $(a, b)$ is in the updated matching $M$ after Step (II), then $(a, b)$ satisfies (3) after Step (III).

There are two possibilities: (i) Prior to the matching update step, the edge $(a, b)$ was in $M$, or (ii) $(a, b)$ is an edge in $M'$ and is added to $M$ by Step (II).

In case (i), the dual weights of $a$ and $b$ remain unchanged in Step (III) and therefore, $(a, b)$ satisfies (3). For case (ii), observe that, at the start of the phase, $(a, b) \in E'$ is a non-matching admissible edge. Therefore, the edge $(a, b)$ satisfies

$$y(a) + y(b) = \bar{c}(u, v) + \varepsilon.$$

The update of (III(b)) does not change the dual weight of $b$, and the update step (III(a)) reduces the dual weight of $a$: $y(a) \leftarrow y(a) - \varepsilon$. After step (III), therefore, we have $(a, b) \in M$ satisfying (3). Thus, at the end of the phase, every edge of $M$ satisfies (3).

Next, we show that, for every edge $(a, b)$ that is both admissible at the start of the iteration and is a non-matching edge at the end of the iteration, $(a, b)$ satisfies (2). There are two possibilities: (i) $b \notin B'$, and, (ii) $b \in B'$.

In case (i), at the start of the phase the edge satisfies (2); note that matching edges at the start of the phase satisfy (3), which means (2) is also satisfied. During the dual update step, only points of $B'$ may undergo a dual update. Since $b \notin B'$, $y(b)$ remains unchanged. The dual weight of $a$ may reduce by $\varepsilon$, but this only increases the slack of $(a, b)$. Therefore (2) continues to hold.

For case (ii), since $(a, b)$ is admissible and remains a non-matching edge at the end of the phase, $(a, b) \notin M'$. Since $M'$ is a maximal matching, $a$ is matched to another vertex $b'$ in $M'$, i.e., $(a, b')$ is in $M'$. While the dual weight of $b$ may increase by $\varepsilon$ in Step (III(b)), step (III(a)) will reduce the dual weight of $a$ (since $a$ is matched in $M'$) by $\varepsilon$, ensuring that the slack on $(a, b)$ remains at least 0. This completes the proof of (I2).

### A.2 Proof of Lemma 3.1

*Proof.* Let $M$ be the matching produced by our algorithm, and let $M_{\text{OPT}}$ be an optimal matching with respect to the cost function $\bar{c}(\cdot,\cdot)$. From (2), and the fact that the dual weights of all free vertices with respect to $M$ are non-negative we have $\sum_{(a,b)\in M} \bar{c}(a,b) = \sum_{(a,b)\in M} y(a)+y(b) \leq \sum_{v\in A\cup B} y(v)$. Note that $M_{\text{OPT}}$ is a perfect matching, and so, from (3), we get $\sum_{v\in A\cup B} y(v) = \sum_{(a,b)\in M_{\text{OPT}}} y(a) + y(b) \leq \sum_{(a,b)\in M_{\text{OPT}}} \bar{c}(a,b) + \varepsilon n$. Combining these two observations completes the proof of the lemma. □

### A.3 Proof of Lemma 3.2

*Proof.* Since the algorithm only increases dual weight magnitudes, it is sufficient to show that the claim holds at the end of the algorithm. First, we show that the claim holds for all vertices of $B$. Let $a$ be an arbitrary free vertex of $A$ at the beginning of the last phase of the algorithm. From invariant (I1), the dual weight $y(a)$ is 0. For every vertex $b \in B$, the edge $(a,b)$ satisfies either equation (2) or equation (3), and, therefore, $y(b) \leq \bar{c}(a,b) + \varepsilon - y(a) = \bar{c}(a,b) + \varepsilon \leq 1 + \varepsilon$. Now, observe that the maximum dual weight magnitude increase for any vertex during a single iteration, including the final iteration, is $\varepsilon$. Thus, the maximum possible value of $y(b)$ for any $b \in B$ at the end of the algorithm is $1 + 2\varepsilon$.

Next, we show that, for any vertex $a \in A$, $|y(a)| < 1 + 2\varepsilon$. If $a$ is free, this holds true from invariant (I1). Otherwise, $a$ is matched to some $b \in B$, and the edge $(a,b)$ is $\varepsilon$-feasible, implying that $y(a) = \bar{c}(a,b) - y(b) \geq -y(b) \geq -(1 + 2\varepsilon)$. Thus, $|y(a)| \leq 1 + 2\varepsilon$ for every $a \in A$. □

### A.4 Proof of Lemma 3.3

*Proof.* Let $b$ be a vertex of $B'$ at the beginning of some iteration. If $b$ remains free at the end of the iteration, then $|y(b)|$ increases by $\varepsilon$ during that iteration. Otherwise, $b$ was matched to a vertex $a$ in $M'$ during this iteration. This implies that $|y(a)|$ increases by $\varepsilon$ during that iteration. Therefore, each vertex of $B'$ causes some vertex to experience an increase to its dual weight magnitude, and the total dual weight increase is at least $\varepsilon n_i$. □

### A.5 Proof of Lemma 3.4

*Proof.* First, we note that the set of free vertices can easily be found in $O(n)$ time at the beginning of each phase. It is easy to see that steps (II) and (III) of the algorithm can be implemented to run in $O(n)$ time, since any matching has size $O(n)$ and each vertex experiences at most one dual adjustment per iteration. It remains to describe how step (I) can be implemented to run in $O(n \times n_i)$ time. In this step, the algorithm computes a maximal matching on the graph $G'(A' \cup B', E')$, where $E'$ is the set of admissible edges with at least one point in $B'$. Note that this graph can be constructed in $O(n \times n_i)$ time by scanning all edges incident on $B'$. Next, the algorithm finds a maximal matching $M'$ in $G'$. This matching $M'$ is found by processing each free vertex $b$ of $B'$ in an arbitrary order. The algorithm attempts to match $b$ by identifying the first edge $(a,b)$ in $G'$ such that $a$ is not already matched in $M'$. If such an edge $(a,b)$ is found, then $(a,b)$ is added to $M'$, and the algorithm processes the next vertex of $B'$. Otherwise, there is no way to add an edge incident on $b$ to $M'$. After all vertices of $B'$ are processed, $M'$ is maximal. Overall, the time for each phase is dominated by the time taken to compute a maximal matching, which is $O(n \times n_i)$. □

### A.6 Proof of Lemma 3.5

*Proof.* Let $M$ be the matching produced by our algorithm, and let $M_{\text{OPT}}$ be an optimal matching with respect to the cost function $\bar{c}(\cdot,\cdot)$. From (2), and the fact that the dual weights of all free vertices with respect to $M$ are non-negative we have $\sum_{(a,b)\in M} \bar{c}(a,b) = \sum_{(a,b)\in M} y(a) + y(b) \leq \sum_{v\in A\cup B} y(v)$. Note that $M_{\text{OPT}}$ is a maximum-cardinality matching, and so all vertices of $B$ are matched in $M_{\text{OPT}}$. Therefore, all vertices unmatched by $M_{\text{OPT}}$ are of type $A$, and the algorithm maintains that such vertices have a non-positive dual weight. Combining this with (3), we get $\sum_{v\in A\cup B} y(v) \leq \sum_{(a,b)\in M_{\text{OPT}}} y(a) + y(b) \leq \sum_{(a,b)\in M_{\text{OPT}}} \bar{c}(a,b) + \varepsilon|B|$. Combining these two observations completes the proof of the lemma. □

## B Algorithm for Optimal Transport

In this section, we extend the push-relabel algorithm described in Section 2.2 to the OT problem. Recall that an instance $\mathcal{I}$ of the OT problem consists of two point sets $A$ and $B$, both consisting of $n$ points, where every point $a \in A$ (resp. $b \in B$) has a demand of $\mu_a$ (resp. supply of $\nu_b$), and the total supply and the total demand both sum to 1. An $\varepsilon$-approximation algorithm for the OT problem outputs a transport plan $\sigma$ that transports all the supply from $B$ to $A$ at a cost $w(\sigma) \leq w(\sigma^*) + \varepsilon$, where $\sigma^*$ is the optimal transport plan with respect to $\mathcal{I}$.

As in the algorithm of Lahn *et al.* [19], our main routine operates under the assumption that all demands, supplies, and edge costs are integers. In order to enforce this assumption, our algorithm transforms the initial OT input $\mathcal{I}$ into a *scaled* instance $\bar{\mathcal{I}}$ where all supplies, demands, and costs are rounded to integers. A solution to $\bar{\mathcal{I}}$ can be mapped back to the original unscaled input $\mathcal{I}$, albeit with some loss of accuracy, due to the rounding.

More precisely, our algorithm will create the scaled instance $\bar{\mathcal{I}}$, with a scaled cost $\bar{c}(a, b)$ on every edge $(a, b) \in A \times B$, a scaled demand $\bar{\mu}_a$ on every $a \in A$, and a scaled supply $\bar{\nu}_b$ on every $b \in B$, as follows: Let $\theta = 6n/\varepsilon$. For every $a \in A$, the algorithm sets $\bar{\mu}_a \leftarrow \lceil \theta \mu_a \rceil$. For every $b \in B$, the algorithms sets $\bar{\nu}_b \leftarrow \lfloor \theta \nu_b \rfloor$. Finally, for every edge $(a, b) \in A \times B$, the algorithm sets $\bar{c}(a, b) \leftarrow \lfloor 4c(a, b)/\varepsilon \rfloor$. For any transport plan $\bar{\sigma}$ with respect to the scaled instance, let $\bar{c}(\bar{\sigma}) = \sum_{(a,b)\in A\times B} \bar{\sigma}(a, b)\bar{c}(a, b)$ be the cost of $\bar{\sigma}$. Let $\bar{\sigma}^*$ be an optimal (i.e., minimum-cost maximum) transport plan with respect to $\bar{\mathcal{I}}$. Let $\mathcal{U} = \sum_{b\in B} \bar{s}_b$ be the total supply with respect to the scaled instance.

In Section B.1, we describe a push-relabel algorithm that computes a transport plan $\bar{\sigma}$ with respect to the scaled instance $\bar{\mathcal{I}}$ such that: (i) $\sigma'$ has at most $n$ unmatched supply, and (ii) $\bar{c}(\bar{\sigma}) \leq \bar{c}(\bar{\sigma}^*) + \varepsilon\mathcal{U}/4$. Such a transport plan $\bar{\sigma}$ can be transformed into a maximum transport plan $\sigma$ with respect to the original instance $\mathcal{I}$; we will refer to this process as *unscaling*. Our algorithm's unscaling process is neary identical to the one presented by Lahn *et al.* [19], and so we will simply summarize the approach and highlight the slight differences.

The unscaling approach begins by setting $\sigma(a, b) \leftarrow \bar{\sigma}(a, b)/\theta$. At this point, there are two issues with $\sigma$: (i) Due to the fact that demands were rounded up during the scaling, each demand vertex $a \in A$ could have a total incoming flow $\sum_{b\in B} \sigma(a, b)$ that exceeds its limit $\mu_a$. (ii) The transport plan $\sigma$ may not be maximum with respect to $\mathcal{I}$. Our algorithm resolves (i) using the following strategy: For any demand vertex $a \in A$, the algorithm repeatedly reduces $\sigma(a, b)$ along an arbitrary edge with $\sigma(a, b) > 0$ until the total flow incoming to $a$ is less than or equal to $\mu_a$. After this, in order to resolve (ii), the algorithm arbitrarily assigns the remaining unmatched supplies to the remaining unmatched demands. Each such supply is matched at an additional cost of at most 1. To bound the total number of supplies matched in this fashion, consider the following three sources:

(a) The rounding down of the supplies when creating $\bar{\mathcal{I}}$ contributes at most $1/\theta$ unused supply per supply vertex, which is $n/\theta$ in total.

(b) The rounding up of the demands when creating $\bar{\mathcal{I}}$ contributes at most $1/\theta$ excess demand. When this excess demand is removed in step (i) above, an additional $1/\theta$ unused supply could be introduced on each supply vertex, which is $n/\theta$ in total.

(c) The algorithm with respect to the scaled instance $\bar{\mathcal{I}}$ does not compute a maximum transport plan; instead it computes a transport plan with at most $n$ unused supplies. After unscaling, these $n$ unused supplies contribute $n/\theta$ to the unmatched supply that is transported arbitrarily by step (ii).

Each of these three sources (a)–(c) contribute at most $n/\theta = \varepsilon/6$ to the unmatched supply prior to step (ii), leading to an additional cost of $\varepsilon/2$. Note that sources (a) and (b) are also present in [19]. However, (c) is not present for their approach because their algorithm on the scaled instance $\bar{\mathcal{I}}$ produces a maximum transport plan, unlike ours, which leaves possibly $n$ unmatched supplies in $\bar{\sigma}$. As a result, they can afford to scale the supplies and demands up by $4n/\varepsilon$, which is smaller than our scaling value of $\theta = 6n/\varepsilon$. Other than this slight difference in the analysis, our scaling process

remains identical to that given by Lahn *et al.*, and so the rest of their analysis for the scaling and unscaling processes can be applied directly to our case as well.[2]

In addition to the error sources (a)–(c) above, there are two additional sources of error:

(d) Since all costs are scaled up by $4/\varepsilon$ and rounded down, the quality of the solution for the scaled instance may be negatively impacted during unscaling.

(e) The solution $\bar{\sigma}$ produced with respect to $\bar{\mathcal{I}}$ is not optimal; specifically, $\bar{c}(\bar{\sigma}) \leq \bar{c}(\bar{\sigma}^*) + \varepsilon\mathcal{U}/4$, where $\bar{\sigma}^*$ is an optimal transport plan with respect to $\bar{\mathcal{I}}$.

Both sources (d) and (e) are also present in the scaling approach of Lahn *et al.* [19]. Following an identical analysis, error sources (d) and (e) contribute an additional additive error of $\varepsilon/2$.[3] Combined with error sources (a)–(c), the total error produced by the algorithm is upper-bounded by $\varepsilon$, as desired. It remains to describe the algorithm for producing the transport plan $\bar{\sigma}$ with respect to the scaled instance $\bar{\mathcal{I}}$.

### B.1 Algorithm for Scaled Instance

For the scaled instance $\bar{\mathcal{I}}$, we are given as input a demand $\bar{\mu}_a$ on each demand vertex $a \in A$, a supply $\bar{\nu}_b$ on each supply vertex $b \in B$, and a cost $\bar{c}(a, b)$ for each edge $(a, b) \in A \times B$. The total supply $\mathcal{U} = \sum_{b \in B} \bar{\nu}_b = O(n/\varepsilon)$. We describe a parallel algorithm that computes a transport plan $\bar{\sigma}$ with respect to $\bar{\mathcal{I}}$ such that $\bar{c}(\bar{\sigma}) \leq \bar{c}(\bar{\sigma}^*) + \varepsilon\mathcal{U}/4$ in $O(\log(n/\varepsilon)/\varepsilon^2)$ time. This is accomplished via reduction to the algorithm for the *unbalanced matching problem*, whose analysis is described in Section 3.3. The input $\bar{\mathcal{I}}'$ for the unbalanced matching problem is constructed from $\bar{\mathcal{I}}$ as follows: Each demand vertex $a \in A$ (resp. supply vertex $b \in B$) is replaced by a set of identical *copy* vertices $\langle a_1, \cdots, a_{\bar{\mu}_a} \rangle$ (resp. $\langle b_1, \cdots, b_{\bar{\nu}_b} \rangle$). Each of these copy vertices is assigned a supply or demand of 1. For every pair of copy vertices $(a_i, b_j)$, $(a, b) \in A \times B$, there is an edge $(a_i, b_j)$ in $\bar{\mathcal{I}}'$ with cost $c(a_i, b_j) = \bar{c}(a, b)$. Let $G'(A' \cup B', A' \times B')$ be the resulting complete bipartite graph for $\bar{\mathcal{I}}'$. We invoke the matching algorithm on $\bar{\mathcal{I}}'$ using a new error parameter of $\varepsilon' = \varepsilon/6$. From Lemma 3.5, the matching algorithm produces a matching $M \subseteq A' \times B'$ with size at least $(1 - \varepsilon/6)|B'|$ and additive error at most $\varepsilon|B'|/6$. Next, we convert $M$ into a transport plan $\bar{\sigma}$. For any edge $(a, b) \in A \times B$, the flow $\bar{\sigma}(a, b)$ is given by the number of matching edges of $M$ between a copy of $a$ and a copy of $b$. The result is a transport plan $\bar{\sigma}$ with at most $\varepsilon\mathcal{U}/6$ additive error. Furthermore, the number of unmatched supplies is at most $\varepsilon|B'|/6 = \varepsilon\mathcal{U}/6 \leq \varepsilon\theta/6 \leq n$ as desired.

To analyze the running time of this algorithm, observe that the matching instance $\bar{\mathcal{I}}'$ has size $O(\Theta) = O(n/\varepsilon)$. Using the running time analysis of the matching algorithm, our OT algorithm runs in $O(n^2/\varepsilon^3)$ sequential time and $O(\log(n/\varepsilon)/\varepsilon^2)$ parallel time.

### B.2 Improved Sequential Algorithm

Although the algorithm of Section B.1 has a fast parallel execution time, the sequential running time of the algorithm is not optimal. In this section, we describe how to speed up the sequential running time of the OT algorithm by a factor of $1/\varepsilon$. To accomplish this, we explicitly maintain the following invariant: We raise the dual weight of any unmatched free supply node $b \in \mathbb{B}$ to be at least as large as the largest dual weight among all its other copies $b' \in \mathbb{B}$. This can be enforced in a straight-forward way while updating the matching. Furthermore, this change in dual weight does not violate any feasibility conditions.

Given this property, like in [20, 30], we show in Lemma B.1 that copies of the same vertex can have no more than two distinct dual weights at any time. Thus, we can maintain at most two clusters for each point and bring down the execution time of each iteration to $O(n^2)$ and the total execution time to $O(n^2/\varepsilon^2)$.

[2] The work of Lahn *et al.* [19] makes use of values $C$ and $\varepsilon$ in their analysis of their scaling and unscaling processes. For the sake of this equivalency, assume without loss of generality that $C = 1$ and $\varepsilon = 0.5$ in their context.

[3] In [19], the scaling of costs is described as a part of the algorithm on $\bar{\mathcal{I}}$ instead of as a part of the initial scaling process that generates $\bar{\mathcal{I}}$. For simplicity in exposition, we present the scaling of costs as a separate step, before the scaled OT algorithm of Section B.1. However, the overall analysis of error sources (d) and (e) remains the same as in [19].

**Lemma B.1.** *At any point in the execution of our algorithm, given three copies of the same vertex $v \in A \cup B$ in $\mathcal{I}'$, at least two of them will share the same dual weight.*

*Proof.* Suppose there are three copies of $v \in B$ that have three distinct dual weights. Let $v_1$ and $v_2$ be two copies of $v$ such that

$$y(v_1) < y(v_2) - 2\varepsilon. \tag{5}$$

From the fact that the dual weight of $v_2$ is larger than that of $v_1$, we conclude that $v_1$ is matched to some $a \in \mathbb{A}$. From (3), we have

$$y(v_1) + y(a) = \overline{c}(a, b). \tag{6}$$

Now consider the edge $(v_2, a)$. From (2), $y(v_2) + y(a) \leq \overline{c}(a, b) + \varepsilon = y(a) + y(v_1) + \varepsilon$. The last equality follows from (6). This implies $y(v_2) \leq y(v_1) + \varepsilon$ contradicting (5). □

**Theorem B.2.** *The push-relabel algorithm can compute an $\varepsilon$-approximation to the optimal transport in $O(n^2/\varepsilon^2)$ sequential time and $O(\log(n/\varepsilon)/\varepsilon^2)$ parallel time.*

# C Experimental Results for Optimal Transport

## C.1 Experimental Results on GPU - Approximation error

In this section, we present the actual approximation error of Sinkhorn and our algorithm with respect to $\varepsilon$ in our experiments in section 5.

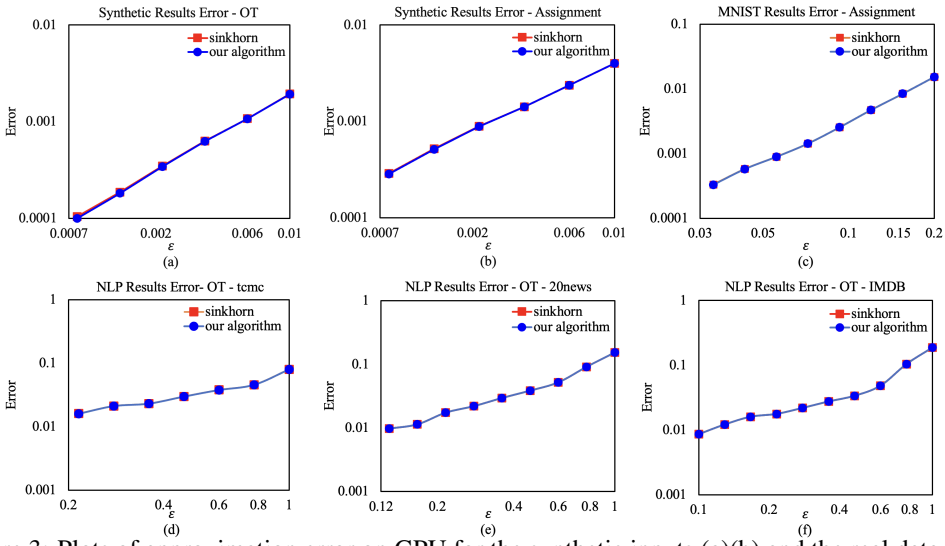


Figure 3: Plots of approximation error on GPU for the synthetic inputs (a)(b) and the real data inputs (c)(d)(e)(f).

## C.2 Experimental Results on GPU - Reverse Process

In this section, we present some additional experimental results that compare our algorithm implementation to the Sinkhorn algorithm. In the main text, for each setup, we generated input data and computed the assignment or OT cost using our algorithm with different values of $\varepsilon$. Then, we determined the appropriate regularization parameter of the Sinkhorn algorithm, ensuring the Sinkhorn distance is close but no lower than the cost of the solution generated by our algorithm. Here we conduct the experiments with the same setup except for the following reversed process: We compute the assignment or OT cost using the Sinkhorn algorithm with different values of the regularization parameter, then we determine the appropriate value of $\varepsilon$ for our algorithm. The results of synthetic and real word data can be seen in Figure 4 and Figure 5.

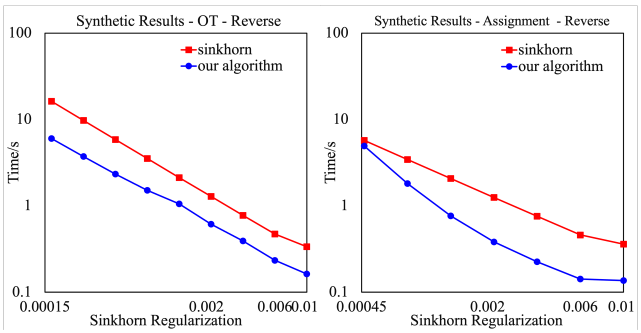


Figure 4: A plot of running times for the synthetic inputs (reverse).

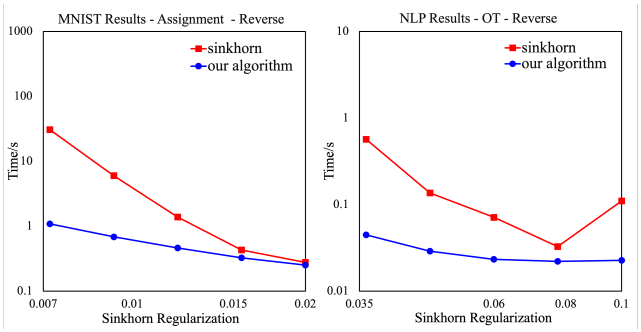


Figure 5: A plot of running times for the real data inputs (reverse).

### C.3 Experimental Results on GPU - Comparison with DROT

In this section, we present an additional comparison against the state-of-the-art algorithm, drot[24], executed on GPU. We note that DROT's implementation does not inherently provide a feasible transport plan. To ensure a fair comparison, we modified their transport plan to be feasible by incorporating an arbitrary mass after getting the output. The rest of the experimental setup and datasets remain consistent with those used in the comparison with Sinkhorn in the main text. The results of these experiments are presented in Figure 6. These results show that our new parallel algorithm consistently performs significantly faster than DROT for both assignment and OT problems.

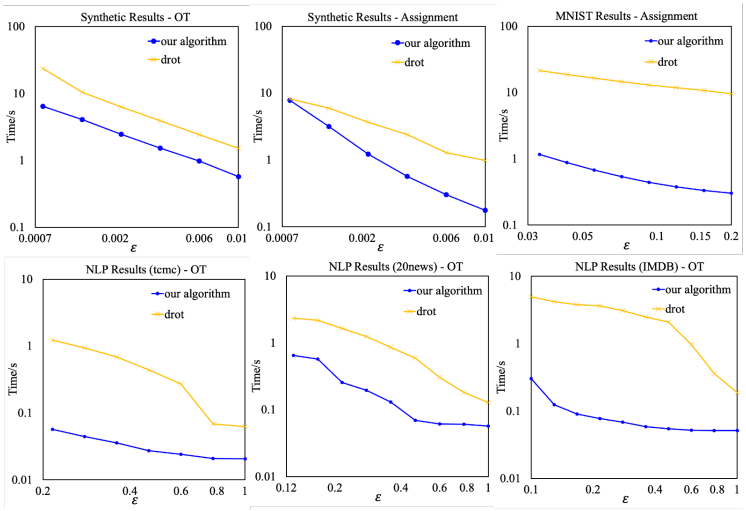


Figure 6: A plot of running times comparison with drot.

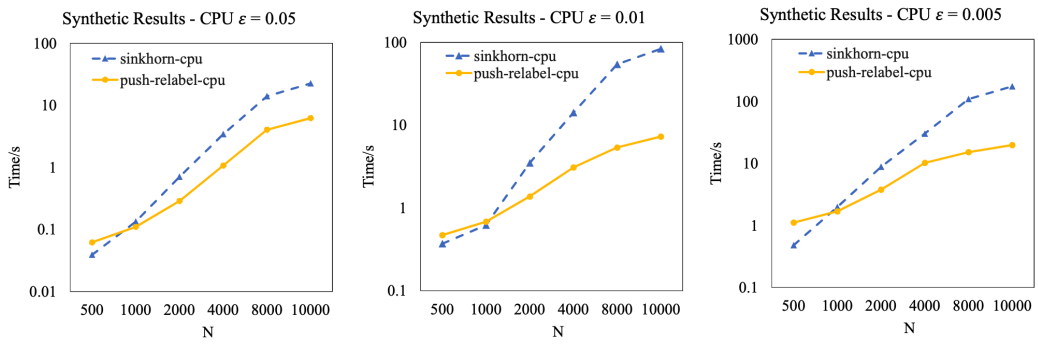


Figure 7: A plot of running times for the synthetic OT inputs.

### C.4 Experimental Results on CPU - Sequential Algorithm Comparison

In this section, we present additional experimental results that compare our sequential implementation to the Sinkhorn algorithm on CPU. Our comparisons are performed on synthetic data sets. Each synthetic input for the OT problem is generated using the following procedure: Each demand vertex $a \in A$ is assigned a demand value $\mu_a$ chosen uniformly at random from the range $[0, 1]$. Afterward, we normalize the demands so that the total demand is 1 by dividing each demand value $\mu_a$ by the total demand $\sum_{a \in A} \mu_a$ from prior to normalization. The supplies $\nu_b$ for all $b \in B$ are randomly assigned using an identical process to that of the demands. Next, we describe how the edge costs for all edges $(a, b) \in A \times B$ are generated. Each vertex of $A \cup B$ is treated as a two-dimensional point, sampled uniformly at random from a unit square. We generate positions for each of the vertex sets $A$ and $B$ by sampling $n$ two-dimensional points uniformly at random from a unit square. For any pair of points $(a, b) \in A \times B$, the cost $c(a, b)$ is set as the Euclidean distance between the points $a$ and $b$.

For each value of $\varepsilon$ in $[0.1, 0.01, 0.005]$, and for each value of $n$ in $[500, 1000, 2000, 4000, 8000, 10000]$, we execute 10 runs. For each run, we invoke both the Sinkhorn algorithm and our algorithm, using both CPU and GPU implementations. For each combination of $n$, $\varepsilon$, and algorithm choice, we average the running times over all 10 runs and record the results. The results can be seen in Figure 7.

When given a particular value of $\varepsilon$ as input, our algorithm may generate a solution with additive error much less than $\varepsilon$. In order to ensure a fair comparison, we run our experiments so that our algorithm and the Sinkhorn algorithm produce a solution of comparable quality. The Sinkhorn algorithm's solution quality is determined by the value of a regularization parameter, which is provided as input to the Sinkhorn algorithm. Within our experiments, for a run on a given value of $\varepsilon$, we first run our algorithm using $\varepsilon$ as input, which produces a transport plan $\sigma$. Next, we perform a binary search over possible values of Sinkhorn's regularization parameter until Sinkhorn generates transport plan $\sigma'$ such that $|c(\sigma) - c(\sigma')| \leq 10^{-6}$. This process was used for all of our experiments.

These results seem to suggest that our CPU-based push-relabel OT algorithm outperforms the CPU-based Sinkhorn algorithm on OT instances.