EFFICIENT, MOSTLY ACCURATE OPTIMAL TRANSPORT

Avigayil Helman
abh2177@columbia.edu

ABSTRACT

Entropic optimal transport is an increasingly popular method for approximating the distances between two distributions. It is used in a range of applications, such as computer vision, computer graphics, generative adversarial networks, and economics. Many variants for solving entropic optimal transport exist, notably the Sinkhorn Algorithm for its fast computational speed [1]. However, the vanilla form of Sinkhorn suffers from numerical instability on close approximations and does not scale well to large datasets. In this paper, we combine the log stabilized epsilon scaling algorithm introduced in [2] with the symbolic matrices of the KeOps library [3] on the GPU for efficient and stable optimal transport. We then compare our results with [1].

1 Introduction

Optimal transport is best described as a way to find the optimal allocation of resources between m producers and n consumers. Each producer holds a share of the total resources to be transferred. Likewise, each consumer can accept a portion of those shares. We can write this as a distribution a and b where:

$$1 = \sum_{i=1}^{m} a_i \tag{1}$$

$$1 = \sum_{i=1}^{n} b_i. \tag{2}$$

Optimal transport attempts to find an allocation plan such that each producer can split its mass across all consumers. Let $P \in \mathbb{R}_+^{mxn}$ be a map of the plan where P_{ij} represents the portion of the resources being transferred from producer i to consumer j and

$$a = P1_n \tag{3}$$

$$b = P^T 1_m. \tag{4}$$

Given a, b, and a cost matrix $C \in \mathbb{R}_+^{mxn}$, we can formulate the optimal transport problem as:

$$\min_P \sum_{i,j} C_{ij} P_{ij} \tag{5}$$

Several algorithms exist for solving (5), such as the auction algorithm or simplex algorithm. These methods can be computationally expensive and hard to solve. A common alternative is to make the problem easier to solve by regularizing the problem, This is done by adding the entropy of P, denoted K, to (5):

$$\min_P \sum_{i,j} C_{ij} P_{ij} - \epsilon K(P) \text{ subject to } \epsilon > 0 \tag{6}$$

$$K(P) = -\sum_{i,j} P_{ij}(log(P_{ij}) - 1) \tag{7}$$

where ϵ is a weighting parameter. Solving (6) is fast for large values of ϵ and easily parallelized.

The addition of the entropy changes the problem by adding a "blur." The larger the blur, the farther the solution to (7) will be from (6). If we minimize the blur by decreasing $epsilon$, our approximation of P will be closer to P^*. However, as is often the case in many optimization problems, as we decrease ϵ our solution can suffer from numerical instability. In cases where a sharp approximation is desired, this presents a problem.

Another common problem that occurs when solving (6), (7) is the polynomial space complexity of storing the cost matrix. This restricts the problem to smaller datasets. To put this into perspective, constructing a cost matrix between two standard sized images of $500x500$ pixels stored as 32 bit floats can eat up $(500x500)x(500x500)*4 = 2.5x10^{11} = 250$ GB of RAM.

The paper is organized as follows: Section 2.1 introduces the Sinkhorn algorithm [1] for solving (7). Section 2.2 reviews a variant of Sinkhorn, the stabilized sparse scaling algorithm [2], which attempts to solve the issues discussed above. Section 2.3 presents symbolic matrices [3][4], a more efficient way of fixing the memory and computational limitations than what is proposed in section 2.2. Lastly, section 3 benchmarks the performance of combining the stability of section 2.2 with symbolic matrices.

2 Background

2.1 Sinkhorn Algorithm

The Sinkhorn algorithm was first applied to entropic optimal transport in [1]. Due to it's fast computational speed, Sinkhorn has since become the default solver for many optimal transport libraries [5].

Sinkhorn is based on the following explicit formula that solves (6):

$$P = diag(u)Kdiag(v) \tag{8}$$

$$K = e^{-\frac{C}{\epsilon}} \tag{9}$$

where K is the kernel matrix and u, v are unknown variables. A proof for (8) can be found in [6].

We can write a, b in terms of (8) as:

$$\begin{aligned} a = P1_n = diag(u)Kdiag(v)1_n = diag(u)Kdiag(v) = u \cdot (Kv) \\ b = P^T 1_m = diag(v)K^T diag(u)1_m = diag(v)K^T diag(u) = v \cdot (K^T u). \end{aligned} \tag{10}$$

And so values for u, v can be found by iterating through the following updates until convergence:

$$\begin{aligned} u^{l+1} = \frac{a}{(Kv^l)} \\ v^{l+1} = \frac{b}{(K^T u^{l+1})}. \end{aligned} \tag{11}$$

These updates are known as the Sinkhorn updates.

2.2 Stabilized Sparse Scaling Algorithm

The stabilized sparse scaling algorithm is an variant of Sinkhorn found in [2]. It proceeds in four steps.

2.2.1 Log Stabilization

The kernel matrix in (9), u, and v can easily run into numerical underflow for small values of ϵ. More than one solution can be found to this problem. [2] attempts to solve this problem while preserving the simple nature of (11) by introducing two more variables α, β. α, β absorb the large values of u, v:

$$\begin{aligned}\alpha &= \alpha + \epsilon \log(u) \\ \beta &= \beta + \epsilon \log(v).\end{aligned} \tag{12}$$

When u, v are absorbed, u, v are reset to the initial values and K becomes:

$$K = e^{-\frac{C+\alpha+\beta}{\epsilon}} \tag{13}$$

and P is becomes:

$$P = diag(e^{\frac{\alpha}{\epsilon}})Kdiag(e^{\frac{\beta}{\epsilon}}) \tag{14}$$

2.2.2 ϵ- scaling

The more blur in the objective function, the faster the Sinkhorn updates converge. That is, the greater the value of ϵ, the faster our algorithm will compute. [2] suggests a modification to the algorithm by starting with a larger ϵ than our target, and reducing ϵ an interval. With this method, we being each successive iteration with a near optimal solution, leading to faster convergence.

[2] recommends starting with a large enough ϵ to prevent numerical instability and decreasing ϵ at a rate of:

$$\epsilon_l = \epsilon_0 \lambda^l. \tag{15}$$

2.2.3 Truncation

Truncation attempts to capitalize on the sparsity of the kernel and coupling matrix for small ϵ to reduce memory consumption. Kernel values less than a certain given parameter are left out of the computation. This reduces overall memory usage and computation.

However, for large ϵ, the kernel matrix remains dense and the overall improvement is minimal.

2.2.4 Multi-Scale

Multi-scaling builds on truncation by implementing a course-to-fine scheme. A hierarchical tree is constructed such that for large ϵ, K and P only iterate over a small subset of the data at the top of the tree. As ϵ is scaled back in the ϵ-scaling, multi-scaling adds successive levels of the tree to K and P. Once the target ϵ is reached, the whole dataset is used.

Since K, P are now smaller for large ϵ, multi-scaling in conjunction with truncation yields better memory and time performance.

2.3 Symbolic Matrices

Symbolic matrices reduce polynomial memory consumption to linear terms by computing values on the spot. Instead of pre-computing an mxn kernel matrix, the kernel matrix can simply be defined as a function. While it requires at least double the time to recompute K each time they are used, symbolic matrices can be parallelized on the GPU for fast performance.

It can be shown that sparse matrices under perform dense matrices when sparsity is greater than 30% [7]. In such cases, this method may be preferred to truncation and multi-scaling.

3 Analysis

3.1 Setup

Our tests were constructed in two parts. First we ran the original open-source implementation detailed in Section 2.2 provided by the author as a baseline [8]. This code is written in C++ using the Eigen library and only provides CPU support. Second, we implemented our own version of optimal transport using log-stabilization, ϵ-scaling, and symbolic matrices. The idea to apply symbolic matrices to the Sinkhorn algorithm was taken from [3] and applied using the Python KeOps library [4] with GPU support.

Our test data consisted of a 2D point cloud in the shape of a heart, circle, line, and swirl. All tests were run on either an Intel Xeon CPu or Nivida SMP provided by Google Collab.

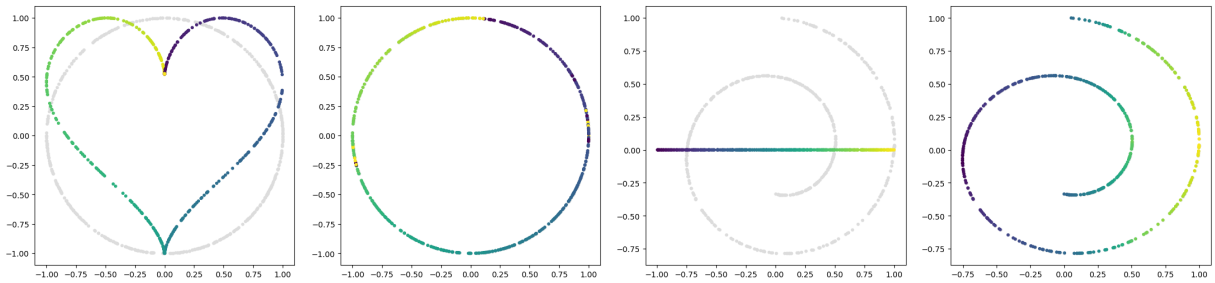

Figure 1: The left two images depict the transport of a heart into a circle. The right two images depict the transport of a line into a swirl. 500 samples were taken for each and an ϵ of 1e-11 was used.

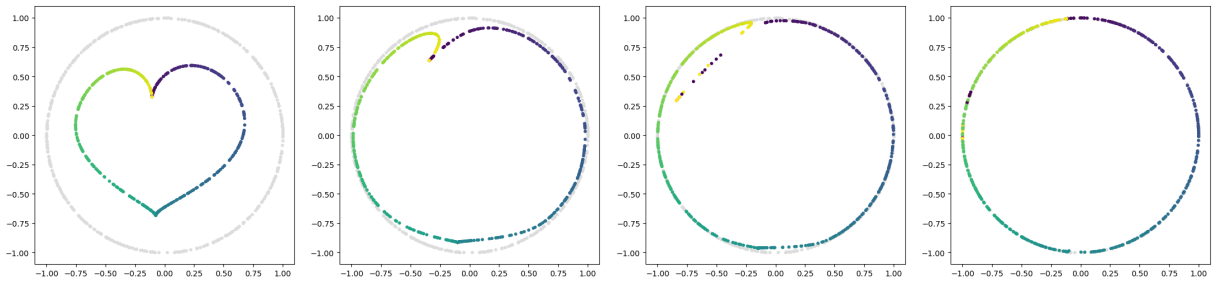

Figure 2: The blur imposed by $\epsilon = [1.0, 0.1, 0.01, 1e-5]$.

3.2 Results

Visual results can be seen in Figure 1 with a precision of $\epsilon = 1e-11$. For both tests, the reported error was below $1e-6$.

We ran tests on both the CPU and GPU code across a range sample sizes using an $\epsilon = 1e-4$. The results can be seen in Figure 2. When running on the CPU using truncation and multi-scaling, our memory ran out above a sample size of 20,000.

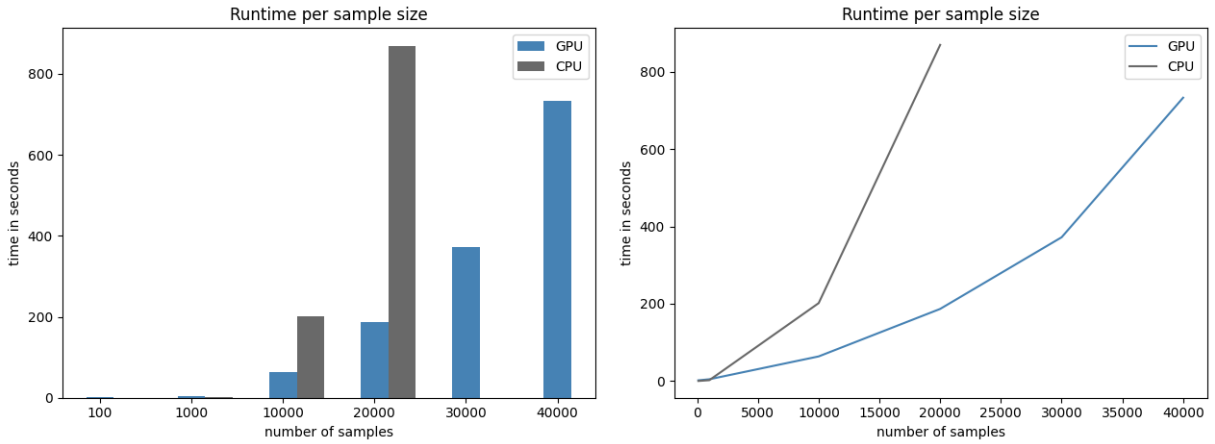

Figure 3: Bar plot and graph for the runtime of both GPU and CPU runs. The CPU data stops at N=20,000 when an out of memory error occurs.

3.3 Discussion

From Figure 1, we can see that it is possible to achieve an almost exact mapping for log stabilization and ϵ-scaling. It goes with out saying, that it is computationally more expensive to reach this level of precision. Figure 2 depicts the blur imposed for different values of ϵ.

Our results show that the GPU version achieves superior time and memory performance. At a sample size of 20,000 our CPU crashed with an out of memory error while our GPU RAM remained steady. Although it is difficult to analyze in Figure 3, initially the CPU does outperform the GPU for sample sizes ≈ 1000. This is largely due to the high cost of transferring data between the CPU and GPU when an kernel is launched. As the number of samples increases, the parallelization speedup of the GPU negates this initial setback.

It is worth noting that there was no documentation for the Multi-scale library and our usage of it may have been incorrect. Additionally, since Google Collab was used for benchmarking our power may have been throttled.

4 Conclusion

Both implementations have their strengths. For small sample sizes such as in Figure 1, the CPU can reach the same accuracy with improved overall performance metrics. For larger sizes, the CPU becomes unfeasible.

Future work includes more extensive and thorough bench-marking than that of Figure 3, performance comparisons of different ϵ sizes, and testing the algorithms on more applications.

References

[1] M. Cuturi, Sinkhorn Distances : Lightspeed Computation of Optimal Transport, Advances in Neural Information Processing Systems (NIPS) 26, 2013

[2] Schmitzer, B. (2016). Stabilized Sparse Scaling Algorithms for Entropy Regularized Transport Problems. arXiv preprint arXiv:1610.06519.

[3] Jean Feydy, Thibault Sejourne, Francois-Xavier Vialard, Shun-ichi Amari, Alain Trouve, and Gabriel Peyre. Inter- ´ polating between optimal transport and mmd using sinkhorn divergences. In The 22nd International Conference on Artificial Intelligence and Statistics, pages 2681–2690, 2019

[4] Charlier, B., Feydy, J., and Glaunès, J. (2018). Kernel operations on the gpu, with autodiff, without memory overflows. http://www.kernel-operations. io. Accessed: 2018-10-04.

[5] Flamary, R., & Courty, N. (2017). POT: Python optimal transport library.

[6] https://optimaltransport.github.io/slides-peyre/CourseOT.pdf

[7] Wang, Ying, and Korhan Cengiz. "Implementation of the Spark technique in a matrix distributed computing algorithm." Journal of Intelligent Systems 31.1 (2022): 660-671.

[8] Schmitzer, B.(2020). MultiScaleOT. https://bernhard-schmitzer.github.io/MultiScaleOT/build/html/index.html

[9] Cisneros, Hugo. https://hugocisneros.com/blog/the-elegance-of-optimal-transport/