Parallelizing t-SNE with CUDA and Python MPI

David Zhang, Valeriy Rotan, Sahil Gupta

Implementations are available at:

- CUDA implementation: https://github.com/Davarco/cs267-project

- MPI implementation: https://github.com/valrotan/tsne-mpi

1 Introduction

t-SNE is an algorithm for visualizing high-dimensional data in a two or three-dimensional space. This is widely applicable across multiple domains; examples of high-dimensional data include raw images, word vector embeddings, or genomics. t-SNE is a nonlinear dimension-reductionality technique focused on capturing the local structure of high-dimensional data better than tools like PCA, which only captures global variance. As a result, t-SNE is well-equipped for datasets where features lie on a complex high-dimensional manifold, as opposed to PCA, which simply captures distance.

However, t-SNE is slow on large and very high-dimensional datasets. The basic algorithm relies on iteratively computing distance matrices between all points in the data, which takes $O(N^2)$ operations each time. In this project, we develop two accelerated implementations of t-SNE and analyze their performance and limitations. First, we develop a CUDA-based implementation of t-SNE and test it on an NVIDIA RTX 2080 TI GPU. We additionally implement an MPI-based implementation of Barnes-Hut t-SNE, building off of the existing scikit-learn implementation. We then compare both implementations to existing CPU implementations in scikit-learn on common datasets such as MNIST.

2 Basic Algorithm

First, t-SNE computes a joint probability distribution, p_{ij}, that represent similarities between datapoints X in the high dimensional space. It does this using the Euclidean distance between point x_i and x_j, such that p_{ij} is higher when the two points are close together and vice-versa.

$$p_{j|i} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2/2\sigma_i^2)}$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

Note that in $p_{j|i}$, distances are converted to probabilities under the assumption that neighboring datapoints are chosen based on a Gaussian centered as x_i. However, this means we must find an appropriate variance of the Gaussian, σ_i, for each datapoint x_i. Intuitively, σ_i should be smaller for dense regions and larger for sparse regions. In practice, we search for σ_i using binary search such that it produces a "perplexity" value specified by the user.

$$\textit{Perplexity}(P_i) = 2^{H(P_i)}$$

$$P_i = -\sum_j p_{j|i} \log_2(p_{j|i})$$

For our lower-dimensional representation of the data, Y, we can also compute a joint probability distribution, q_{ij}. The goal of t-SNE is to find Y such that q_{ij} mimics p_{ij}. To do so, we compute a

cost function C, which is the Kullback-Leibler divergence between distributions P and Q. Then, we iteratively run gradient descent, which is given below.

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$$

2.1 Barnes-Hut t-SNE

The gradient

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$$

can be interpreted as a collection of all forces exerted on a point in the low-dimensional space by all other points in the low-dimensional space. As such, if the points are in a stable configuration, it can be interpreted as having found the minimization for the Kullback-Leibler divergence between the high-dimensional and low-dimensional similarity distributions. The gradient can be decomposed into the sum of attractive forces and repulsive forces. While attractive forces can be efficiently computed, repulsive forces require an $O(n^2)$ search. Therefore, the Barnes-Hut approximation algorithm can be used for the repulsive forces, bringing the complexity down to $O(n \log n)$.

3 Methods and Approach

3.1 CUDA

3.1.1 Similarities in High-Dimensional Space

To compute p_{ij} in a parallel fashion, we first compute the distance matrix D_x between all points in X. Rather than compute $||x_i - x_j||^2$ directly, we use the following expansion.

$$||x_i - x_j||^2 = ||x_i||^2 + ||x_j||^2 - 2x_i^T x_j$$

Then, when we compute the distance matrix, we can do

$$X_{norm} = \begin{bmatrix} ||x_1||^2 & ... & ||x_n||^2 \end{bmatrix}^T$$

$$D_x = \texttt{repeat}(X_{norm}, \texttt{dim=0}) + \texttt{repeat}(X_{norm}, \texttt{dim=1}) - 2X^T X$$

To compute X_{norm}, we first write a kernel to square each entry of X. Then, we sum over each row using the thrust library (reduce_by_key).

To compute D_x, we write kernels to duplicate X_{norm} both row-wise and column-wise, and sum them together. We use the GEMM function in cuBLAS to compute $X^T X$.

Next, we must fit σ_i for each datapoint. We use a combination of custom kernels and thrust to compute $p_{j|i}$, entropy $H(P_i)$, and update σ_i in parallel. To simplify our code, we don't use a stopping condition for binary search, instead opting to run 50 search iterations for all datapoints.

3.1.2 Gradient Descent on Low-Dimensional Representation

First, we generate our initial points Y by sampling random numbers from a uniform distribution on the CPU. Then, we run gradient descent for a fixed number of iterations.

Before computing the gradient, we first find q_{ij}, which is done similarly to p_{ij} above. We write a kernel to compute each term within the summation of the gradient, across all i of $\frac{\delta C}{\delta y_i}$ in parallel. Then, we use thrust to perform the summation.

For stability, practical implementations of t-SNE also re-center Y around 0 at each iteration of gradient descent. We compute the mean of all y_i by multiplying Y with a vector of ones using GEMM in cuBLAS. While we would've preferred to use thrust for performance, the reduce_by_key function didn't support summations over columns easily.

3.2 Python MPI

To evaluate the scaling performance of Barnes-Hut t-SNE with MPI, we build off of the scikit-learn implementation and use MPI for Python. We choose to mainly optimize the iterative gradient descent step, as this is the biggest bottleneck in computing t-SNE.

3.2.1 Gradient Descent on Low Dimensional Representation

The gradient computation step in barnes-hut tsne is split into two parts. First, a quadtree is constructed using the current iteration's points in low-dimensional space. Then, the quadtree is used to compute the positive and negative forces for each of the points. We keep the computation of the quadtree on a single process, and parallelize the gradient computation and application steps. When each process has a copy of the quadtree, the gradients can be computed and applied independently, with the exception of a few aggregates that need to be shared between all the processors. We additionally use async MPI calls where applicable to reduce the overhead of communication between processes.

4 Results

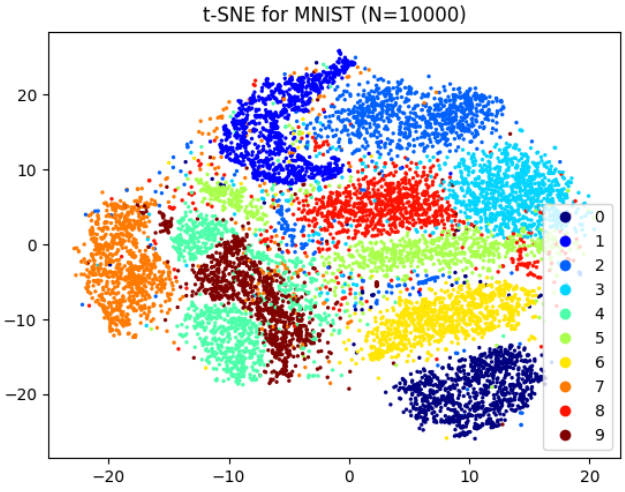

Figure 1: Visualization of 10000 MNIST data points with our implementation.

Figure 1 shows an example result of our CUDA implementation on the MNIST dataset. All runs are done on the raw high-dimensional space; in MNIST, each datapoint is represented by 784 numbers (28x28 image). We do not use PCA to first reduce it to a smaller dimension, which is done by default in libraries like scikit-learn.

Each color in the visualization represents a different digit (0-9). Even after reducing the image to a two-dimensional space, we find that datapoints of the same digit are still clustered together. This confirms that t-SNE is able to keep the local structure of high-dimensional data.

4.1 Performance

Figure 2 shows the performance comparison of the most commonly used t-SNE implementations (sklearn's exact and barnes-hut implementations) as well as our implementations. For all evaluations, we used the MNIST dataset with all 784 features. We set perplexity to 30, used 250 exaggeration epochs with a factor of 4, and set the number of output components to 2.

Both, our CUDA and MPI implementations were able to beat the baselines. Sklearn's exact implementation uses the basic $O(N^2)$ version of the algorithm, and as such rapidly explodes in computation

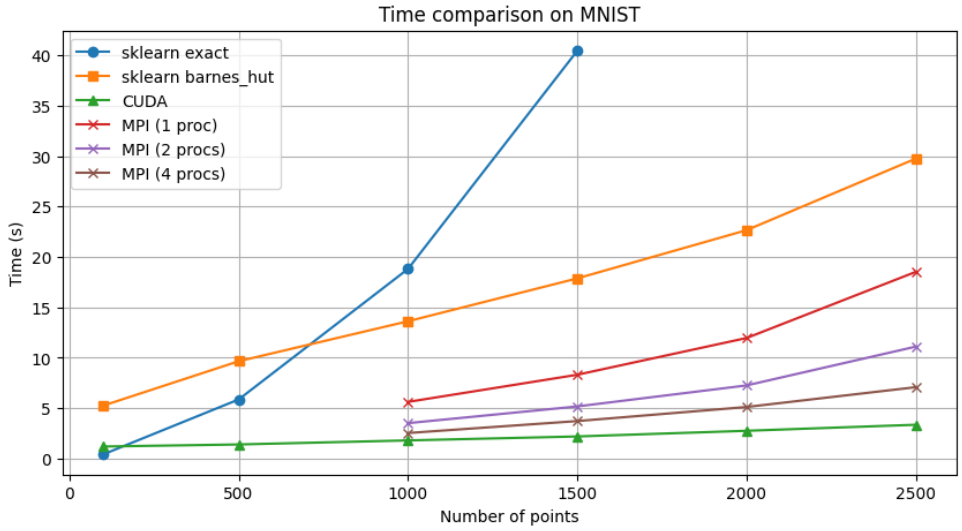

Figure 2: Performance plots of the tested algorithms

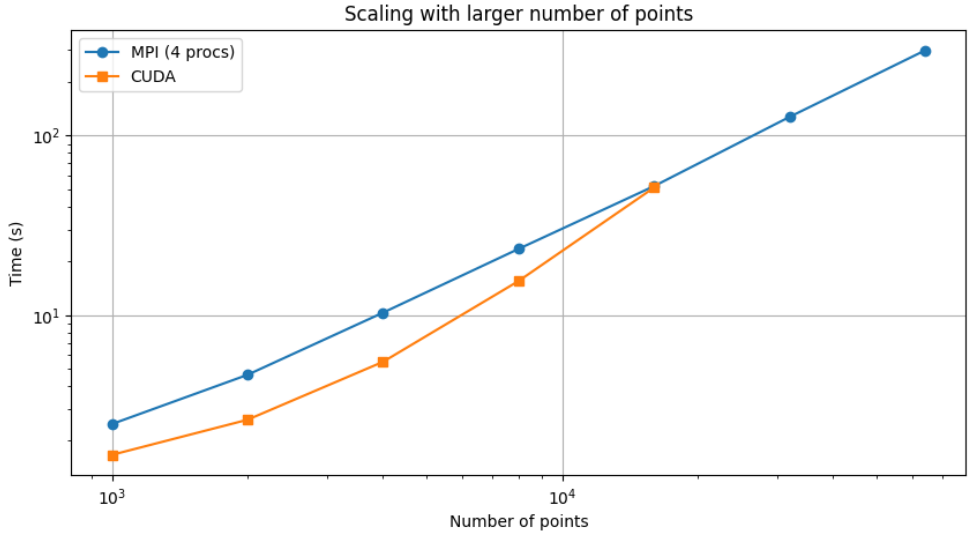

Figure 3: Performance plots on larger number of points for selected algorithms

time beyond 500 points. The CUDA implementation also uses the $O(N^2)$ algorithm, however because of the efficient GPU kernels and matrix operations, the CUDA implementation still outperforms other implementations below 2500 points. The MPI implementations outperform the sklearn implementation, but are behind the CUDA implementations under 2500 points. As we go from 1 to 2 processes, our implementation gets a 1.67x speedup, and 2.62x speedup when using 4 processes instead of 1.

Figure 3 shows performance of MPI with 4 cores and CUDA when using larger numbers of points. All settings remain as before, but we now evaluate using more points. As we can see, the MPI implementation begins to match or outperform the CUDA implementation at 16000 points. The reason for this is that the CUDA uses the naive $O(N^2)$ implementation which benefits greatly from optimized GPU kernels, while the MPI implementation extends the Barnes-Hut algorithm. Additionally, the MPI implementation works successfully on 32k and 64k points, and can scale to

more points efficiently. The CUDA implementation is $O(N^2)$ in memory as well as compute, and is thus limited by the GPU memory. We use an 11GB RTX 2080 Ti for all our experiments. Our theoretical maximum number of points for the GPU implementation is somewhere between 16k and 32k points. This is a significant limitation for the CUDA implementation, and future work could focus on figuring out how to take advantage of the GPU when using the Barnes-Hut version of the algorithm, as it contains a lot fewer regular matrix operations.

5 Reflection and Future Work

To summarize, we hypothesized that the t-SNE algorithm could be faster using parallel programming techniques. To test this, we implemented a version of t-SNE using CUDA and MPI and benchmarked them against existing CPU implementations in scikit-learn. We found that our MPI version achieves modest speedups relative to scikit-learn, while the GPU-based one is much faster. However, in practice, we found that the GPU is strongly bottlenecked by memory storage, which helps explain why regular t-SNE GPU implementations aren't used in practice. In particular, the distance matrix requires us to store $O(N^2)$ numbers at once, which limits us to running t-SNE on less than 50000 datapoints. This can be more easily avoided in the CPU by using a Barnes-Hut version, increasing memory, or adding more swap space, though the number of datapoints is still limited.

We found that writing code in pure CUDA can be quite cumbersome, which inspired us to use cuBLAS. The syntax for cuBLAS took a while to get used to, especially since matrices are stored as column-order. In retrospect, cuDNN might have been helpful for our implementation, though we were unfamiliar with the library at the time.

References

L. Dalcin and Y. -L. L. Fang, "mpi4py: Status Update After 12 Years of Development," in Computing in Science Engineering, vol. 23, no. 4, pp. 47-54, 1 July-Aug. 2021, doi: 10.1109/MCSE.2021.3083216.

Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, Alejandro Cosimo, Parallel distributed computing using Python, Advances in Water Resources, Volume 34, Issue 9, 2011, Pages 1124-1139, ISSN 0309-1708, https://doi.org/10.1016/j.advwatres.2011.04.013.

Lisandro Dalcín, Rodrigo Paz, Mario Storti, MPI for Python, Journal of Parallel and Distributed Computing, Volume 65, Issue 9, 2005, Pages 1108-1115, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2005.03.010.

Lisandro Dalcín, Rodrigo Paz, Mario Storti, Jorge D'Elía, MPI for Python: Performance improvements and MPI-2 extensions, Journal of Parallel and Distributed Computing, Volume 68, Issue 5, 2008, Pages 655-662, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2007.09.005.

https://distill.pub/2016/misread-tsne/

L.J.P. van der Maaten and G.E. Hinton. Visualizing data using t-SNE. Journal of Machine Learning Research, 9(Nov):2431–2456, 2008.

Laurens van der Maaten. (2013). Barnes-Hut-SNE.