Parallelizing Gradient Calculations in t-SNE

Vy Nguyen
Carnegie Mellon University
Pittsburgh, PA 15213
vtnguyen@andrew.cmu.edu

Jason Zhang
Carnegie Mellon University
Pittsburgh, PA 15213
jasonz3@andrew.cmu.edu

Abstract

In this work, we parallelize t-distributed Stochastic Neighbor Embedding (t-SNE), a non-linear dimensionality reduction method, using CUDA on GPUs from the GHC machines. Modern implementations of t-SNE such as Scikit-Learn use nearest neighbor and Barnes-Hut approximations to reduce algorithmic complexity. By exploiting several degrees of parallelism, we are able to produce high-quality 2D visualizations of the MNIST dataset with more than $2.38\times$ speedup over Scikit-Learn's vectorized CPU implementation.

1 Introduction

Data visualization is an essential tool in data analysis, especially with the emergence of high-dimensional data in modern machine learning tasks. Typically data visualization is achieved by creating a mapping from $\mathbb{R}^d$ to $\mathbb{R}^2$ or $\mathbb{R}^3$ where d is the dimensionality of the original dataset. t-SNE is an especially powerful visualization technique since it attempts to preserve local distances in its embeddings (van der Maaten and Hinton [2008]). Given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$ with $x_i \in \mathbb{R}^d$ for all i, t-SNE aims to output a set of 2- or 3-dimensional vectors $\{y_1, y_2, \ldots, y_N\}$. In this work, we will assume $y_i \in \mathbb{R}^2$ for all i. t-SNE models a similarity distribution p_{ij} given by Equation 1 over all pairs x_i, x_j in the high-dimensional space and a corresponding distribution q_{ij} given by Equation 2 for the low-dimensional space.

$$p_{ij} = \frac{\exp(-\|x_i - x_j\|_2^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|_2^2/2\sigma_i^2)} \tag{1}$$

$$q_{ij} = \frac{(1 + \|y_i - y_j\|_2^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|_2^2)^{-1}} \tag{2}$$

In order to preserve local relationships between vectors in $\mathcal{D}$ in the low-dimensional space, t-SNE trains $\{y_1, y_2, \ldots, y_N\}$ using gradient descent to minimize the KL-divergence between p_{ij} and q_{ij}. An exact implementation of this algorithm has algorithmic complexity $O(N^2)$ per gradient descent iteration since the gradient for each point considers contributions from all other points. However, van der Maaten [2014] shows that the gradient calculations can be interpreted as an N-body problem, and with two well-justified approximations the per-iteration complexity can be driven down to $O(N \log N)$. In particular, the gradient $\frac{\partial \mathcal{L}}{\partial y_i}$ where $\mathcal{L}$ is the KL-divergence can be expressed by Equation 3 where $Z = \sum_{k \neq l}(1 + \|y_k - y_l\|_2^2)^{-1}$.

$$\frac{\partial \mathcal{L}}{\partial y_i} = 4\left(\sum_{j \neq i} p_{ij} q_{ij} Z(y_i - y_j) - \sum_{j \neq i} q_{ij}^2 Z(y_i - y_j)\right) \tag{3}$$

The first term in Equation 3 can be interpreted as attractive forces between all pairs of y_i, y_j. Since p_{ij} decays exponentially for increasingly distant x_i, x_j, a nearest neighbors approximation is valid.

Specifically if $\mathcal{N}_i^k$ is the set of k indices j with the k smallest values of p_{ij}, the attractive forces may be approximated by Equation 4.

$$\sum_{j \neq i} p_{ij} q_{ij} Z(y_i - y_j) \approx \sum_{j \in \mathcal{N}_i^k} p_{ij} q_{ij} Z(y_i - y_j) \tag{4}$$

Evaluating the attractive forces for one point is now $O(kN) = O(N)$ for small constant k. Analogously, the second term in Equation 3 can be interpreted as repulsive forces between all pairs of y_i, y_j. Unlike the attractive forces, the repulsive forces may not be approximated using nearest neighbors due to the heavy-tailed t-distribution q_{ij}. Instead, van der Maaten [2014] introduces a Barnes-Hut approximation where the embedding points $\{y_1, y_2, \ldots, y_N\}$ are placed into a quad-tree. A quad-tree node n_i has the following attributes where $y_{j,1}$ and $y_{j,2}$ are the first and second components of y_j respectively:

- A bounding box spanning $[\alpha_i^-, \alpha_i^+]$ for the first dimension and $[\beta_i^-, \beta_i^+]$ for the second dimension. Let $\bar{\alpha}_i = (\alpha_i^- + \alpha_i^+)/2$ and $\bar{\beta}_i = (\beta_i^- + \beta_i^+)/2$.
- A set of embedding points $\mathcal{S}_i = \{y_j : \alpha_i^- \leq y_{j,1} \leq \alpha_i^+, \beta_i^- \leq y_{j,2} \leq \beta_i^+\}$ contained within the bounding box.
- A center of mass $\bar{y}_i$ corresponding to the average of all $y_j \in \mathcal{S}_i$.
- Four child nodes that partition its bounding box into four equal boxes: $[[\alpha_i^-, \bar{\alpha}_i], [\beta_i^-, \bar{\beta}_i]], [[\bar{\alpha}_i, \alpha_i^+], [\beta_i^-, \bar{\beta}_i]], [[\alpha_i^-, \bar{\alpha}_i], [\bar{\beta}_i, \beta_i^+]], [[\bar{\alpha}_i, \alpha_i^+], [\bar{\beta}_i, \beta_i^+]]$

Suppose for a point y_j and a node n_i that $\|y_j - \bar{y}_i\|_2$ is large. The Barnes-Hut approximation allows us to compute the repulsive forces from the points in $\mathcal{S}_i$ on y_j as if all points in $\mathcal{S}_i$ are concentrated together at $\hat{y}_i$. This is expressed by Equation 5 where we use $q_{jk}^2 Z = (1 + \|y_j - y_k\|_2^2)^{-2}/Z$.

$$\sum_{k \in \mathcal{S}_i} q_{jk}^2 Z(y_j - y_k) \approx \frac{|\mathcal{S}_i|(1 + \|y_j - \bar{y}_i\|_2^2)^{-2}}{Z}(y_j - \bar{y}_i) \tag{5}$$

The Barnes-Hut approximation allows us to calculate the repulsive forces on y_j in $O(\log N)$ time by traversing the quad-tree and applying Equation 5 to distant nodes. With the nearest neighbors and Barnes-Hut approximations, we summarize the t-SNE algorithm below.

Algorithm 1: $O(N \log N)$-per iteration approximation of t-SNE

```
Compute the k-nearest neighbors N_i^k for each x_i
Compute the sparse p_ij matrix for nearest neighbor pairs
Randomly initialize {y_1, y_2, ..., y_N}
for t ∈ {1, ..., T} do
    Compute the attractive forces for each y_i: Σ_{j∈N_i^k} p_ij q_ij Z(y_i − y_j)
    Construct a quad-tree with the points {y_1, y_2, ..., y_N}
    Compute the repulsive forces for each y_i by traversing the quad-tree
    Combine the attractive and repulsive forces to compute ∂L/∂y_i for all y_i
    Update {y_1, y_2, ..., y_N} using gradient descent
end
```

Algorithm 1 has many opportunities for parallelism, but the heaviest is computing the repulsive forces. The quad-tree is the key data structure in this algorithm since each iteration of gradient descent requires (1) construction of a new quad-tree based on the updated $\{y_1, y_2, \ldots, y_N\}$ and (2) Barnes-Hut traversal for each of the N embedding points to calculate the repulsive forces. We focus our efforts on parallelizing these two steps of the t-SNE algorithm.

Constructing a quad-tree in parallel is difficult since the parallelism is dynamic and hidden within the nodes of the quad-tree: for a particular node, the points contained within it can be assigned to the child nodes in parallel. On the other hand, traversing the quad-tree has clear data-parallelism: one can simply launch N threads to traverse the quad-tree and calculate the repulsive forces independently. However having one thread per point creates significant read contention on global memory, and because the embedding points move positions across iterations, the workload distribution is not static.

Aside from the repulsive forces, most of the remaining pieces of the t-SNE algorithm have clear data-parallelism (computing the attractive forces and applying gradient updates in particular). We note that the nearest-neighbor search performed at the beginning is a difficult problem to parallelize. Furthermore, initializing p_{ij} in Equation 1 is also non-trivial in practice since the convention is to initialize each σ_i^2 using binary search so that the log-entropy of p_{ij} over j has a particular value (van der Maaten and Hinton [2008]). However, these steps are not the focus of this work.

2 Approach

As mentioned above, computing the embedding gradients has two components: attractive and repulsive forces. The attractive forces require identifying nearest neighbors for each point before gradient descent, but during gradient descent, the neighbor points are fixed so there is a high degree of memory locality for the attractive forces. This makes the attractive forces trivial to parallelize. Computing the nearest neighbors is a difficult problem in its own, but we can leverage fast similarity search engines such as FAISS (Johnson et al. [2021]) to precompute the nearest neighbors before gradient descent. Additionally, we refer to code from Chan et al. [2018][1] for parallel initialization of the p_{ij} matrix via binary search.

We concentrate our efforts on the repulsive forces since they much harder to compute, requiring the embedding points to be organized in a quad-tree as per the Barnes-Hut algorithm. Our implementation focuses on parallelizing quad-tree construction, leveraging the power of dynamic parallelism in CUDA. We also arrive at an optimized computation of the center of mass through backward traversal of the quad-tree. Furthermore, in order to minimize divergence among CUDA threads, we try multiple approaches for tree traversal, before deciding on using thread-local stacks to perform depth-first search (DFS) traversal.

2.1 Parallel Construction of Quad-Tree through Dynamic Parallelism

An efficient implementation of quad-tree is very essential in the algorithm pipeline, as embedded points move and a new tree has to be constructed after every iteration. A naïve, sequential implementation of quad-tree would take $O(N \log N)$, which would become a bottleneck in the pipeline. Kelly and Breslow [2010] and NVIDIA CUDA Advanced Topics[2] introduce an array-based linear tree representation that utilizes bucket sorting and exclusive scan to parallelize the building of the quad-tree. Essentially, we construct a quad-tree layer-by-layer starting from the root, and a block of threads is launched for each node in a layer. This construction algorithm is highly optimized for GPU and CUDA, and is able to give us roughly 100x speedup on tree construction in both Iris and MNIST dataset relative to a sequential tree construction.

2.1.1 Array-tree representation

To utilize data locality, instead of having node addresses spread out in memory, we organize them in a contiguous array that follows a heap-like structure for traversal. Given a node index i in the tree, its four children are at indices $4i + 1$, $4i + 2$, $4i + 3$, and $4i + 4$, corresponding to the top left, top right, bottom left, and bottom right quadrants of the parent's bounding box respectively. More specifically, layer 1 of the quad tree will take up indices 0 in the quad tree array, layer 2 indices $1 - 4$, layer 3 indices $5 - 20$, and so on. Due to this relative arrangement, for each node, it only takes $O(1)$ time to find its children's indices, as well as its direct parent's index.

2.1.2 Bucket Sort and Exclusive Scan

Beyond the array tree representation, instead of inserting each embedding point-by-point to build the quad tree, we use the idea of bucket sorting to construct the tree in parallel. In particular, we can think of the original, unsorted array of embedding points as the root node representation of the tree. To build its four children, we spawn a kernel grid of one block to bucket sort the unsorted array into regions corresponding to the four quadrants of the parent's bounding box.

[1] `https://github.com/CannyLab/tsne-cuda`

[2] `https://developer.download.nvidia.com/compute/DevZone/C/html_x64/CUDA_Advanced_Topics.html`

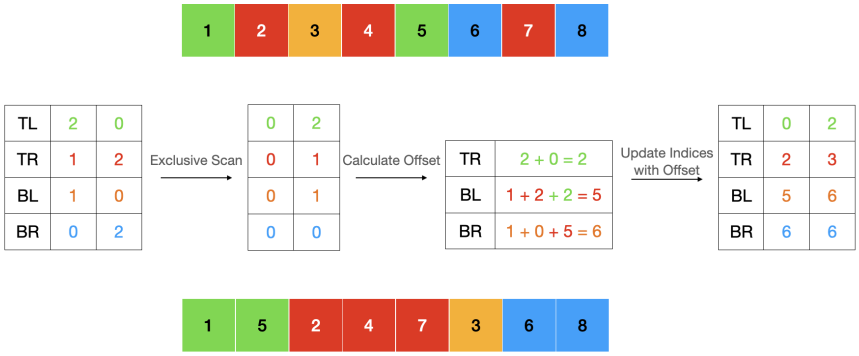


Figure 1: CUDA quad tree construction demonstration. Assume that there are 2 threads, with thread 1 responsible for nodes 1, 3, 5, 7 and thread 2 responsible for 2, 4, 6, 8. Notice how the updated indices for each bucket in thread 1 (0,2,5,6) correspond to the starting indices for each quadrant in the new sorted array

To bucket sort the array, assuming there are `num_threads` threads in a block, we first allocate a 2D shared memory integer array `bucket_counts` of dimension $4\times$ `num_threads`, in which `bucket_counts[0, j]` corresponds to the total number of top left quadrant points counted by thread j, `bucket_counts[1, j]` corresponds to the top right quadrant, and so on.

We then assign unsorted embedding points to block threads in an interleaved manner to maximize data locality, and have each thread count the number of points per bucket and store their counts in their corresponding indices in the shared memory array. Afterwards, we perform exclusive scan on each row of `bucket_counts` to calculate the offsets of each bucket with respect to the new sorted array. (We utilize the warp-level exclusive scan implementation from Assignment 2.) Finally, we have each thread iterate over their previously assigned points and populate those points into the new array given the calculated offsets. Figure 1 demonstrates the details of this algorithm.

After all the points of a node in level i have been sorted, we update its children's `start` and `end` indices, as well as set their bounding boxes. Next, we repeat the same process on the four children by launching a kernel grid consisting of four blocks, and each block is responsible for a sorted quadrant. It is clear that this is a recursive procedure, and we leverage the power of dynamic parallelism in CUDA help us achieve optimal run time in quad tree construction.

2.1.3 CUDA Dynamic Parallelism

To implement this construction algorithm, we use CUDA dynamic parallelism, in which a parent kernel can launch a child kernel. After completing bucket sort, we launch a new grid with 4 blocks corresponding to the four children of the parent kernel's node, each responsible for performing the recursive bucket sorting on its smaller set of points. The recursion ends at the base case, when a node is given only one point, or when the depth of the quad-tree has exceeded the allocated depth.

2.2 Efficient Computation of Center of Mass

A drawback of this quad-tree implementation is that we do not have a way of efficiently computing the center of mass for each node during construction time without going through all the points in its number range in a linear fashion. Such a naïve computation of the center of mass would destroy the speedup we gain from the tree construction. Thus, we implement a backward computation of the center of mass, starting from the final layer, and working back up the tree. Although we only compute center of mass after the quad-tree construction has been finished, this computation adds little overhead to our running time (this will be shown in the next section).

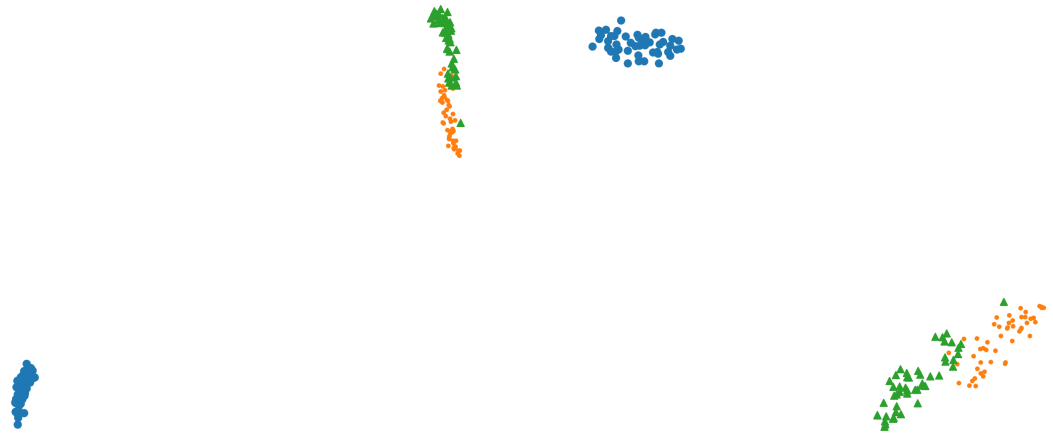
Since we specify a max depth for our quad tree and organize the nodes formulaically, it is not difficult to identify the range of nodes in the final layer. Layer d of the quad tree array will have nodes in the range from $\frac{4^d-1}{3}$ inclusive to $\frac{4^{d+1}-1}{3}$ exclusive. Our implementation starts from the final layer of the

quad tree, where the center of mass is trivial to compute, and work our way up. During construction time, we only populate the center of masses for base case nodes, initializing all others to be zero. There are only 2 base cases: (1) when a node only has one point, in which case the center of mass is simply the point itself, or (2) when a node is at the final layer and has multiple points due to maximum depth constraint, in which case the center of mass is the average of all the points in that node. For nodes higher up the quad tree, the center of mass is simply the weighted average of its four children's center of masses.

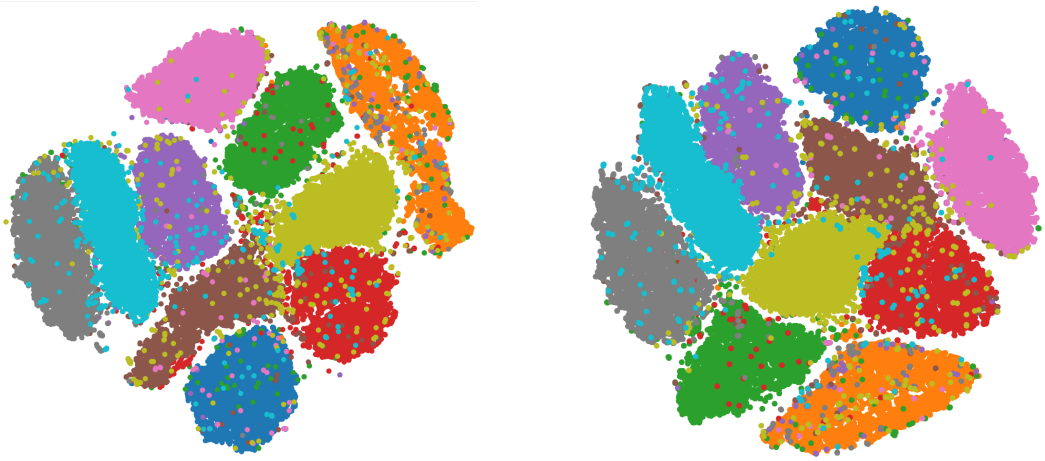
2.3 Tree Traversal Using Thread-Local Stacks

At first, we implemented tree traversal using recursive depth-first search. This traversal technique was a bottleneck in our pipeline since deep recursive calls on the call stack creates high overhead in CUDA. We switched to a non-recursive stack-based implementation to reduce this overhead and exploit thread-local memory.

3 Results and Discussion



(a) Our Embeddings (left) versus Scikit-Learn Embeddings (right) of Iris



(b) Our Embeddings (left) versus Scikit-Learn Embeddings (right) of MNIST

Figure 2: Barnes Hut t-SNE visualization of two datasets: Iris (top), and MNIST (bottom)

To evaluate our performance, we compare our t-SNE model against Scikit-Learn, which implements a vectorized sequential version of Barnes-Hut t-SNE for CPU. We reproduce our generated embeddings for Iris and MNIST datasets along with Scikit-Learn's embeddings in Figure 2. Visually, our

embedding plots show clear clusters for the three classes in Iris and the ten classes in MNIST. To quantify the quality of these embeddings, we apply k-means clustering on these embeddings and record the clustering accuracies. These measurements along with run-times are shown in Table 1.

Experiment	Clustering Accuracy	Run-time (sec)	Speed-up
Iris (Scikit-Learn)	0.9133	0.43	-
Iris (Ours)	0.9333	1.00	0.43x
MNIST (Scikit-Learn)	0.7883	343.28	-
MNIST (Ours)	0.7435	143.88	2.38x

Table 1: Run-times and clustering accuracies for Scikit-Learn and our t-SNE implementations on Iris and MNIST. Clustering accuracy is measured by taking the mode of each cluster as the label.

From Table 1, we can see that k-means is able to achieve reasonably good accuracies just from fitting to our embeddings. Although our reported clustering accuracy on MNIST is lower than Scikit-Learn's, the random initialization in t-SNE causes variation across different runs, so we have no reason to believe there is statistical difference in clustering accuracy between our implementation and Scikit-Learn's. Furthermore, we can see that **our implementation is slower than Scikit-Learn on Iris but 2.38x faster on MNIST.** We note our reported speed-ups are also under-estimates since we were not able to run FAISS on CUDA (the GHC machines did not have cuBLAS, and we were not able to use a GPU from PSC). In particular for our MNIST experiment, FAISS on CPU consumes 20.37 seconds of the 143.88 seconds.

The reason our implementation observes speedup on MNIST but not on Iris is due to problem scale. Iris has 150 data points whereas MNIST has 60,000. In general, we expect our t-SNE implementation to yield higher speedups for larger datasets. Our implementation builds a full quad-tree on GPU, requiring several recursive kernel-launches. This overhead becomes significant on Iris when there are only 150 points and traversal is takes very little time. On the other hand, the number of data points for MNIST is much larger so its workload is dominated by traversal, allowing our parallelizations to manifest speedups. This can be verified in Figure 3 where we see over 60% of the execution time on Iris is spent building the quad-tree whereas more than 80% of the execution time on MNIST is spent traversing the quad-tree.

While we achieve speedup over Scikit-Learn on MNIST, our speedup is not as good as we initially anticipated. Other works that cite large speedups over Scikit-Learn such as Chan et al. [2018] reference Scikit-Learn's exact $O(N^2)$ t-SNE implementation as a baseline whereas we take Scikit-Learn's Barnes-Hut version as our baseline. These figures are what led us to anticipate much higher speedups in our proposal. As mentioned previously, **we observe slowdown on Iris because the number of points in the dataset is so small that our execution time is dominated by overhead from our quad-tree construction.** Scikit-Learn[3] uses a sequential point-by-point tree construction which is much faster for a small number of points. On MNIST, **our speedup is limited by read contention on the quad-tree in global memory**. The vast majority of execution time is spent traversing the quad-tree, and in our best configuration, we spawn one thread for each of the 60,000 data points in MNIST. With this many threads reading the quad-tree irregularly from global memory, we are certain that our speedup is limited by memory contention. Our speedup on MNIST may be higher if we could bring in portions of the quad-tree in shared memory.

We attempted to mitigate global memory contention in the traversal step by reducing the number of blocks/threads and having each thread process multiple points; however, these configurations were even slower likely due to poor load balancing. Load balancing is difficult in our quad-tree traversal since there is no way to reliably find points that require similar execution times for repulsive force calculation. We attempt to bucket sort the points at each iteration and perform interleaved thread assignment so threads get points within a similar region; however, nearby points can be at totally different depths in the tree, and in our experiments, load balancing did not offer any additional speedup. This is further complicated by the movement of embedding points across iterations into clusters. In early iterations, embedding points are evenly spread out, and the constructed trees are thus more balanced compared to later iterations. As t-SNE converges, embedding points can get very dense and clustered, causing the quad tree to be very deep and large. As a result, our tree traversal

[3] `https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/manifold`

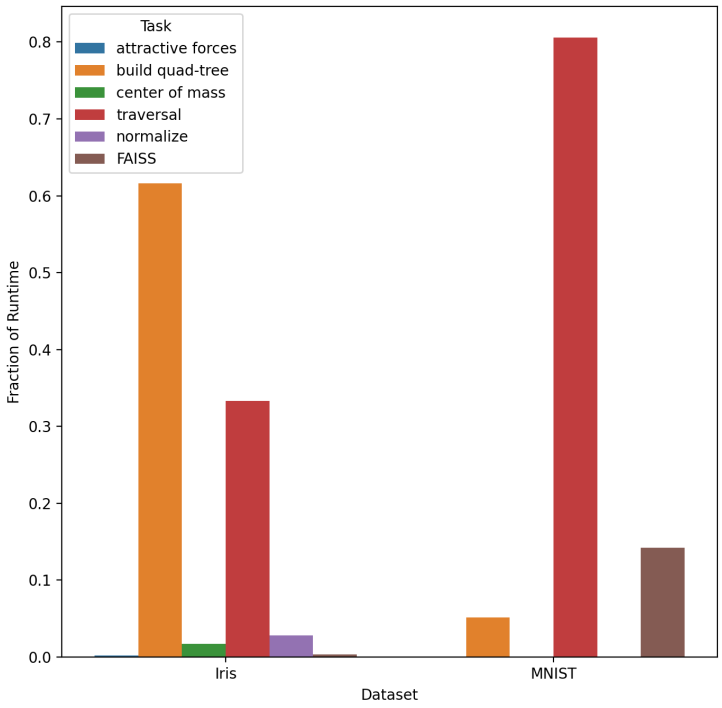


Figure 3: Breakdown of execution time for our t-SNE implementation on Iris vs. MNIST

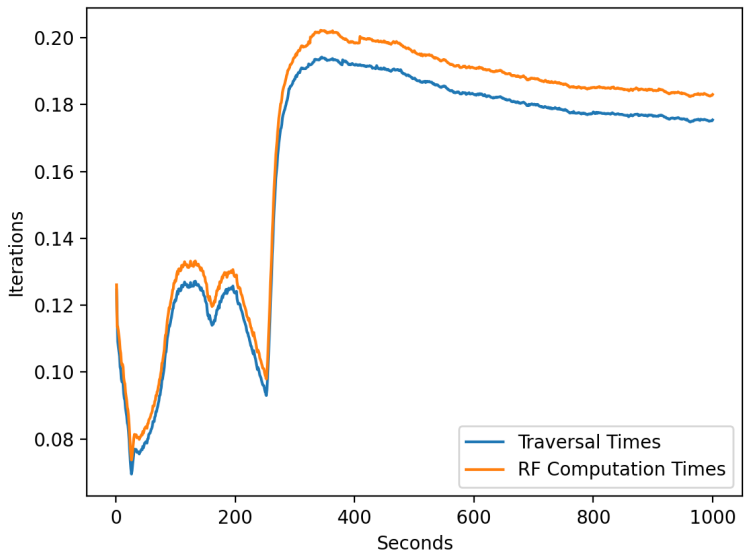


Figure 4: Repulsive forces computation times and tree traversal times through 1000 iterations on MNIST. Following van der Maaten and Hinton [2008], for the first 250 iterations t-SNE trains with early exaggeration, amplifying p_{ij}'s by a factor of 12. Thus causes fluctuation in execution times for repulsive forces in earlier iterations. However once the clusters start to form after 250 iterations, traversal times stabilize at a higher value.

and repulsive forces computations get much slower over iterations. Figure 4 clearly demonstrates this phenomenon.

4 Individual Contributions

Both of us contribute equally to the project. Jason implemented the data loaders, parallelized quad-tree construction, experimented with load-balancing in the traversal step, and evaluated clustering accuracies of the embeddings. Vy implemented the skeleton of the t-SNE algorithm, gradient calculations, non-recursive quad-tree traversal, and generated the embedding visualizations.

References

David M. Chan, Roshan Rao, Forrest Huang, and John F. Canny. T-sne-cuda: Gpu-accelerated t-sne and its applications to modern data. *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sep 2018. doi: 10.1109/cahpc.2018.8645912. URL `http://dx.doi.org/10.1109/CAHPC.2018.8645912`.

Jeff Johnson, Matthijs Douze, and Herve Jegou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, Jul 2021. ISSN 2372-2096. doi: 10.1109/tbdata.2019.2921572. URL `http://dx.doi.org/10.1109/tbdata.2019.2921572`.

Maria D. Kelly and Alex D. Breslow. Quadtree construction on the gpu : A hybrid cpu-gpu approach. 2010.

Laurens van der Maaten. Accelerating t-sne using tree-based algorithms. *Journal of Machine Learning Research*, 15(93):3221–3245, 2014. URL `http://jmlr.org/papers/v15/vandermaaten14a.html`.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL `http://www.jmlr.org/papers/v9/vandermaaten08a.html`.