t-SNE-CUDA: GPU-Accelerated t-SNE and its Applications to Modern Data

David M. Chan[†*], Roshan Rao[†‡], Forrest Huang[†§] and John F. Canny[¶]
EECS Department, University of California, Berkeley
Berkeley, CA, USA
Email: [*]davidchan@berkeley.edu, [‡]roshan_rao@berkeley.edu, [§]forrest_huang@berkeley.edu, [¶]canny@berkeley.edu

***Abstract*—Modern datasets and models are notoriously difficult to explore and analyze due to their inherent high dimensionality and massive numbers of samples. Existing visualization methods which employ dimensionality reduction to two or three dimensions are often inefficient and/or ineffective for these datasets. This paper introduces t-SNE-CUDA, a GPU-accelerated implementation of t-distributed Symmetric Neighbour Embedding (t-SNE) for visualizing datasets and models. t-SNE-CUDA significantly outperforms current implementations with 50-700x speedups on the CIFAR-10 and MNIST datasets. These speedups enable, for the first time, visualization of the neural network activations on the entire ImageNet dataset - a feat that was previously computationally intractable. We also demonstrate visualization performance in the NLP domain by visualizing the GloVe embedding vectors. From these visualizations, we can draw interesting conclusions about using the L2 metric in these embedding spaces. t-SNE-CUDA is publicly available at `https://github.com/CannyLab/tsne-cuda`.**

***Index Terms*—Artificial intelligence, Machine learning, Projection algorithms, Dimensionality Reduction, t-SNE, CUDA**

I. Introduction

The recent emergence of large-scale, high-dimensional datasets has been a major factor contributing to advances in the areas of Machine Learning and Artificial Intelligence. While researchers have developed numerous methods for visualizing medium-sized data-sets, such visualizations are often inefficient or ineffective for high-dimensional or large-scale data. This leads to major bottlenecks in a data scientist's research pipeline. Because developing conceptual understandings of the global and local structures of these datasets is vital for successfully developing and improving models, we introduce a fully GPU-based implementation of t-Distributed Stochastic Neighbor Embedding which will allow researchers to explore structure in high-dimensional data efficiently and reduce the burden of forming understandings of the data and models in modern day machine learning tasks.

t-Distributed Stochastic Neighbor Embedding (t-SNE) [15] is a dimensionality-reduction method that has recently gained traction in the deep learning community for visualizing model activations and original features of datasets. t-SNE attempts to preserve the local structure of data by matching pairwise similarity distributions in both the higher-dimensional original data space and the lower-dimensional projected space. As opposed to PCA and sub-sampling which both reduce the signal in the data, and hence the quality of the visualization, t-SNE has been shown to generate interesting low-dimensional clusters of data faithful to the distributions in the original data space [15]. Unfortunately, current t-SNE implementations are inefficient for visualizing large-scale datasets. All current publicly available implementations executes on the CPU and can require large amounts of time to operate on even modest-sized data (the fastest implementations take over 10 minutes to compute the embedding of the 50,000-image CIFAR-10 dataset), running t-SNE on larger datasets can be intractable.

In this work, we introduce t-SNE-CUDA, an optimized implementation of the t-SNE algorithm on the GPU. By taking advantage of the natural parallelism in the algorithm, as well as techniques designed for computing the n-body problem, t-SNE-CUDA scales the t-SNE algorithm to large-scale vision datasets such as ImageNet [3]. Our contributions are as follows:

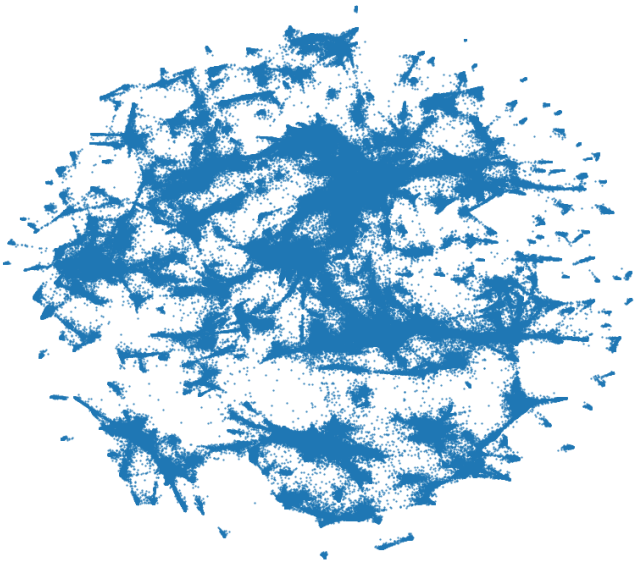

Fig. 1: Clustering of the ResNet-200 codes (2048 dimensional) on all 1.2M ImageNet dataset. This embedding was computed in 486s using an NVIDIA Titan X GPU, the same amount of time required to compute the embedding of the MNIST dataset using current state-of-the-art methods.

† Denotes equal contribution among authors

- We describe the implementation details of t-SNE-CUDA, our publicly available optimized t-SNE implementation using the Barnes-Hut method and approximate nearest neighbours techniques. t-SNE-CUDA significantly outperforms existing methods with a 50-700x speedup without significantly impacting cluster quality.
- We compare and contrast visualizations of real-world large-scale datasets and models, and present some insights into them that can be gleaned from running t-SNE on real-world scale data.

II. Related Work

t-distributed Symmetric Neighbouring Embedding (t-SNE) [15] is widely used in prior work among researchers to visualize data in computer vision and other domains. Van der Maaten *et al.* qualitatively evaluated t-SNE's performance on both MNIST and CIFAR-10 in [15]. DeCAF [4], DeVise [5] and other tools [7] all use t-SNE to help understand the activation space of deep convolutional networks. In addition, t-SNE has been used to aid in visualization and understanding of spatio-temporal video data [27], [30]. In many of these works, the analysis was restricted by the efficiency of t-SNE, and thus researches could only analyze subsets of the data or projections of the data into smaller spaces. Our work allows for complete visualizations at the scale required by these papers.

Current popular implementations of the t-SNE algorithms use tree-based algorithms and approximate nearest neighbours to optimize t-SNE. BH-TSNE [29] and Multicore-TSNE [28] use the Barnes-hut method to approximate repulsive forces during the training process of t-SNE to reduce computational complexity. Pezzotti *et al.* [19] use a forest of randomized Kd trees to compute approximate nearest neighbors for the t-SNE algorithm in a steerable manner to emphasize points users deemed important. While the code presented in [19] can be fast, it does not scale to high dimensional data due to the curse of dimensionality and requires very coarse approximations to achieve significant speedups. Section IV shows that t-SNE-CUDA clearly outperforms existing publicly available methods by large factors, while maintaining a very high level of accuracy.

In addition to t-SNE, other methods have been explored by data scientists and vision researchers to visualize high dimensional data such as Sammon Mapping [24], Isomap [26], Locally Linear Embedding [21], Randomized Principle Component Analysis [20] and Johnson-Lindenstrauss Embedding [12]. In general, t-SNE has been shown to better preserve local structures and similarity between data points compared to these methods. We redirect interested readers to Van der Maaten's and Arora *et al.* 's works [1], [15] for a thorough comparison between t-SNE and these visualization methods.

One potential application for fast t-SNE is low-latency, interactive visualization of neural networks. Such visualizations have been shown by [23] to increase the productivity of data scientists, and several previous works have explored using t-SNE for such active and interactive visualization. [18] suggests using t-SNE for progressive visual analysis of deep neural networks, while [16] suggests t-SNE as a method for increasing user involvement in the training process of DNNs. While we do not explore applications of t-SNE-CUDA to this field of visualization, we believe that it is intriguing and exciting future work, as t-SNE-CUDA is fast enough to visualize training-time embeddings in real-time.

III. Methods

A. t-SNE

t-distributed Symmetric Neighbour Embedding (t-SNE) [15] is a dimensionality reduction method that reduces high dimensional data into a low dimensional embedding space for primarily visualization applications. t-SNE computes the distribution of pairwise similarities in the high dimensional data space, and attempts to optimize visualization in a low dimensional space by matching the distributions using KL divergence. t-SNE models pairwise similarities between points in both higher and lower dimensional space as conditional probabilities $p_{j|i}$ and $q_{j|i}$. The conditional probability $p_{j|i}$, for instance, can be interpreted as the probability that a point j is a neighbor of point i in the higher dimensional space. t-SNE models the probabilities as a Gaussian distribution around each data points in the higher dimensional space,

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

and models the target distribution of pairwise similarities in the lower dimensional embedding space using a Student's t-distribution around each data point to overcome the overcrowding problem in the Gaussian distribution:

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

t-SNE then minimizes the KL divergence between the distributions, which conserves the local structure of data points across the higher and lower dimensional spaces.

To minimize the KL divergence, t-SNE employs gradient descent with the gradient computed as follows:

$$\frac{\partial C}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})q_{ij}(\mathbf{y}_i - \mathbf{y}_j) \tag{3}$$

$$Z = \sum_{k\neq l}(1 + ||y_k - y_l||^2)^{-1} \tag{4}$$

with p_{ij}(and similarly, q_{ij}) computed as the symmetrized joint probabilities of $p_{i|j}$ and $p_{j|i}$ such that $p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N}$.

The t-SNE gradient computation can be reformulated as an N-body simulation problem by rearranging the terms into attractive forces and repulsive forces:

$$F_{attr} = \sum_{j\in[1,\ldots,N], j\neq i} p_{ij}q_{ij}Z(\mathbf{y}_i - \mathbf{y}_j) \tag{5}$$

$$F_{rep} = -\sum_{j\in[1,\ldots,N], j\neq i} q_{ij}^2 Z(\mathbf{y}_i - \mathbf{y}_j) \tag{6}$$

$$\frac{\partial C}{\partial y_i} = 4(F_{attr} + F_{rep}) \tag{7}$$

B. Attractive Forces

For the attractive forces, the term p_{ij} in (5) decreases exponentially with squared distance between the points x_i, x_j in the higher dimensional space. If we take $p_{ij} = 0$ beyond some threshold distance, this introduces significant sparsity into the attractive force calculation. In practice, instead of defining a threshold distance, the K nearest neighbors of each point are obtained and $p_{j|i}$ for points outside the K nearest neighbors of i is taken to be zero. As symmetrizing $p_{j|i}$ can at most double the number of nonzero values, p_{ij} can be stored as a sparse matrix with at most $N*2K$ nonzero entries instead of N^2 nonzero entries. By iterating over only nonzero values of p_{ij}, the attractive force computation becomes linear in N. Empirically, using relatively small values of K (between 32 and 150) are sufficient to achieve reasonable visualizations.

There are a number of methods for computing the k-Nearest Neighbors of a point. The most common implementation is the "Kd Tree" which partitions the search space using the nodes of a tree. While the Kd tree is able to find all nearest neighbors in $O(dN\log(N))$ time (where d is the number of dimensions), this is only sufficient for low dimensional queries. It is computationally expensive for higher dimensions, which are common in many modern data-analysis problems as the dimension dominates the computational cost. In general because $d >> K$, we end up finding the exact nearest neighbors in $\approx O(KN)$ time which is no better than a naive search, whereas better performance can be achieved for finding *approximate* nearest neighbors.

To solve the approximate nearest neighbor problem, Pezzotti *et al.* [19] use a random forest of approximate K-d trees, which allowed them to compute the nearest neighbors in a faster manner, however their approach still suffers from high dimension. There are, however, a number of modern techniques for approximate neighbor selection in high dimension that further reduces the computational complexity. t-SNE-CUDA uses the FAISS [9] library, which provides an efficient, easy to use GPU implementation of similarity search. FAISS is designed for "Billion scale data," and allows us to scale our library to very large datasets.

FAISS is based on locally-sensitive hashing around Voronoi cells in the data. It uses the IDFVAC indexing structure presented in [8] as an indexing structure. Database vectors y are encoded using

$$y \approx q(y) = q_1(y_+ q2(y - q_1(y)) \tag{8}$$

where q_1 and q_2 are quantizing functions. The q_1 function is a coarse quantizer, while the q_2 quantizer is a more fine approximation encoding the residual value. We then rephrase the nearest neighbor problem

$$\mathcal{N}_x = k\text{-argmin}_{y\in N}||x - y_i|| \tag{9}$$

as an approximate asymmetric distance problem. First, the algorithm compute

$$\mathcal{N}_x^{IVF} = \tau\text{-argmin}_{c\in\mathcal{C}}||x - c|| \tag{10}$$

This gives a coarse grained approximation of the location of the point x in terms of "centroids" in $\mathcal{C}$. We then construct our nearest neighbors

$$\mathcal{N}_§ \approx k\text{-argmin}_{y\in N|q_1(y)\in\mathcal{N}_x^{IVF}}||x - q(y)|| \tag{11}$$

By storing the index as an inverted file, and grouping the vectors around the centroids, we can achieve a look-up by linearly scanning $O(\tau)$ inverted lists.

For our implementation, we choose $|C| = \sqrt{N}$, and train the vectors C using the k-Means algorithm. Thus, q_1 is the id of the nearest centroid. q_2 is much more precise, and is selected using product quantization [8] which interprets the vector y as a set of quantized sub-vectors. For more details of this algorithm, we refer interested readers to [9]. The parameter τ, selected by the user, controls the accuracy of the KNN algorithm.

Once the k-Nearest Neighbors are computed, we can compute a sparse matrix P_{ij} which stores the nonzero values of p_{ij}. The attractive force can then be computed efficiently by decomposing it as a series of matrix operations. Let Q_{ij} represent the matrix of q_{ij} values, and Y represent the $N \times 2$ matrix of points in the lower dimensional space. Additionally, let O be a $N\times 2$ matrix of ones and $\odot$ represent the Hadamard product of two matrices. We first distribute the multiplication of F_{attr} giving,

$$F_{attr} = 4Ny_i \sum_j p_{ij}q_{ij} - 4N\sum_j p_{ij}q_{ij}y_j \tag{12}$$

$$F_{attr} = 4N((P_{ij} \odot Q_{ij})O \odot Y - (P_{ij} \odot Q_{ij})Y) \tag{13}$$

Since $P \odot Q$ is computed only once, this becomes one matrix-matrix subtraction, two Hadamard products, and two matrix-matrix multiplications. To achieve the $O(NK)$ run-time, we represent P_{ij} as a sparse matrix with a nonzero value at i, j iff j is a neighbor of i. Q_{ij} is never computed in its entirety; instead, the matrix $P \odot Q$ is computed directly by iterating over nonzero values of P. The matrix-matrix multiplications are performed using cuSPARSE.

C. Repulsive Forces

The repulsive force is more challenging to approximate because the long tails of the Student's T-distribution create reasonably strong repulsive forces even at intermediate distances. This is where the Barnes-Hut approximation occurs. At each iteration, the lower dimensional points $y_1, \ldots, y_N$ are placed in a quad tree. Then, for each point a depth-first-search is performed on the quad tree. When looking at a quad tree cell centered at y_{cell} with radius r_{cell}, the following condition is evaluated:

$$\frac{r_{cell}}{||y_i - y_{cell}||} < \theta \tag{14}$$

If this evaluates to true, then the cell is deemed far enough away to be used as a summary of the forces for all children and the recursion halts. θ is a parameter that controls the accuracy of the approximation with $\theta = 0$ giving the $O(N^2)$ algorithm.

Algorithm 1: General T-SNE-CUDA Algorithm

Input: $N \times d-$dimensional array of data
Output: $N \times 2-$dimensional projection

```
FAISS Computation (approximate k-NN)
Use pairwise distances to compute sparse matrix of P_ij
for i = 1 to convergence do
    R-Force Tree Building (build tree for Barnes-Hut)
    R-Force Computation (use tree to compute approximate repulsive forces)
    Compute P_ij ⊙ Q_ij
    A-Force cuSPARSE (sparse matrix times dense vector)
    Apply Forces (apply forces to points in lower dimensional space)
end for
return lower dimensional projection
```

If a cell is deemed far enough away, the force on point y_i is given by

$$\frac{N_{cell}(y_i - y_{cell})}{(1 + ||y_i - y_{cell}||^2)^2} \approx \sum_{j \in cell} q_{ij}^2 Z^2 (y_i - y_j) \qquad (15)$$

Note that unlike in (6), here the normalization constant Z is squared. However, the normalization constant can also be approximated by simultaneously computing an approximate reduction over q_{ij}.

Given this formula, the approximation is performed in 5 steps: 1) Compute a bounding box of the points, 2) Build a hierarchical decomposition by inserting points into a quad tree, 3) Compute the number of points in each internal cell, 4) Sort points by spatial distance, 5) Compute forces on points using the quad tree.

The code that t-SNE-CUDA uses for computing these steps is adapted to fit the t-SNE objective from [2]. [2] provides an implementation of GPU tree construction and traversal that attempts to minimize thread divergence, wait times, and other sources of slowdowns on the GPU.

D. Algorithm

Algorithm 1 gives the full outline of the discussed sections. Our full implementation is publicly available, so we omit many of the code details, and reserve space in the paper for a mathematical overview of the algorithm, and a discussion of the performance.

IV. Performance

In this section we discuss the performance of our algorithm through the lens of some real-world empirical experiments.

A. Experiments

1) Target Environment: We perform experiments given in this paper using a system with an Intel i7-5820K Processor, containing 6 physical cores (12 with hyper threading) and 64GB of DDR4 RAM. The GPU in use on this system is the NVIDIA Titan-X Maxwell edition GPU, with 3072 CUDA cores clocked at 1.0 GHz and 12GB of GDDR5 memory. It supports a maximum memory speed of 7.0Gbps, with a maximum memory bandwidth of 336.5Gbps. The GM200 chip (Titan-X) has 3072Kb of L2 cache, and 24 Streaming Multiprocessors (6GPCs), with a theoretical peak performance of 6.12TFLOPs for single precision floating point operations. The CPU platform has a theoretical peak performance of 691.2GFlops, giving a peak-to-peak theoretical margin of 8.85x. The CUDA grid sizes have been optimized according to our unique GPU, and for each kernel using a grid search across a set of synthetic problems - For brevity, we provide the full details of implementation as well as optimized grid sizes in our online repository.

2) Datasets: **Simulated Data:** It is important to be able to benchmark methods in a controlled experiment, so we construct simulated data consisting of equal-sized clusters of points sampled from four high-dimensional Gaussian distributions. In these experiments, we are able to vary the size and dimensions of the data, and effectively examine the performance of different algorithms in a controlled environment.
MNIST: The MNIST dataset [14] is a classic computer vision dataset consisting of 60,000 training images, and 10,000 testing images depicting different handwritten digits (numerals 0-9). Each of these digits is black and white, with dimensions 28x28 constituting a 784 dimensional image space.
CIFAR: The CIFAR datasets [11] both consist of a 50,000 image subset from the larger tiny-image dataset. The datasets have images of 10 (Resp. 100) classes such as "ship", "car", "horse", "frog" etc. The images from the CIFAR-10/100 datasets are full color images with dimensions of 32x32x3, giving a 3072 dimensional data space to explore.

B. Synthetic Data

Figure 2 compares the running time of our algorithm with existing implementations on a synthetic dataset that consists of various number of Gaussian-distributed data points for 50 dimensions in 4 clusters. We can see from the Figure that in general we are one to two orders of magnitude faster than even the best current implementations. At 32,000 points, we achieve a 346x speedup over SkLearn with 50 dimensions, and a 86x speedup over Multicore t-SNE. At 512,000 points we achieve a speedup of 1946.92x over SkLearn and a 459.13x speedup over Multicore t-SNE. We also experimented with six and eight core variants of the Multicore t-SNE algorithm, however both had worse or similar performance to the four core variant (MULTICORE-4) of the algorithm.

C. Real-World Datasets

Figure 3 and 4 shows the time taken by our algorithm to compute the embeddings for the MNIST and CIFAR datasets and the speedup compared to current state-of-the-art CPU implementations. t-SNE-CUDA significantly outperforms the popular SKLearn toolkit with more than 700 times speedup over the CIFAR-10 dataset, and with more than 650 times speedup over the MNIST dataset. t-SNE-CUDA also achieves more than 50 times speedup over the state-of-the-art implementations in both datasets.

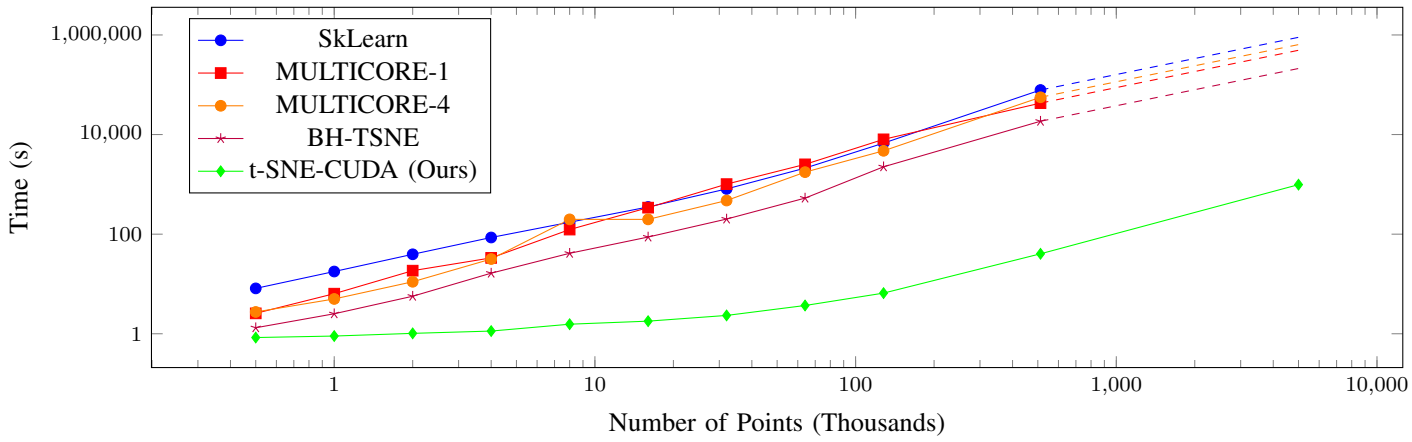

Fig. 2: Time taken compared to other state-of-the-art algorithms on synthetic datasets with 50 dimensions and four clusters for varying numbers of points. Note the log scale on both the x and time axis, and that the scale of the x axis is in thousands of points (thus, the values on the x axis range from 0.5K to 10M points). Dashed lines represent projected times. Projected scaling assumes an $O(n \log n)$ implementation. For small numbers of points, the GPU is not fully saturated leading to better than $O(n \log n)$ scaling. When the GPU becomes fully engaged, our algorithm exhibits a clear $O(n \log n)$ scaling pattern.

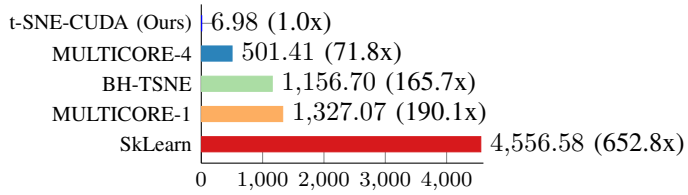

Fig. 3: The performance of t-SNE-CUDA compared to other state-of-the-art implementations on the MNIST dataset. t-SNE-CUDA runs on the raw pixels of the MNIST dataset (60000 images x 768 dimensions) in under 7 seconds.

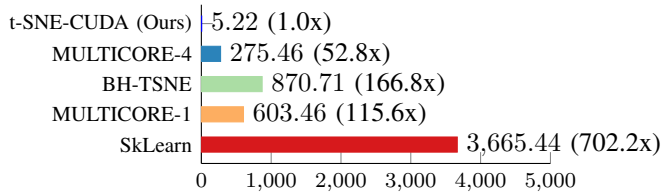

Fig. 4: The performance of t-SNE-CUDA compared to other state-of-the-art implementations on the CIFAR-10 dataset. t-SNE-CUDA runs on the output of a classifier on the CIFAR-10 training set (50000 images x 1024 dimensions) in under 6 seconds. While we can run on the full pixel set in under 12 seconds, Euclidean distance is a poor metric in raw pixel space leading to poor quality embeddings.

Fig. 5: Comparison of different clustering techniques on the MNIST dataset in pixel space. Left: MULTICORE-4 (501s), Middle: BH-TSNE (1156s), Right: t-SNE-CUDA (Ours, 6.98s)

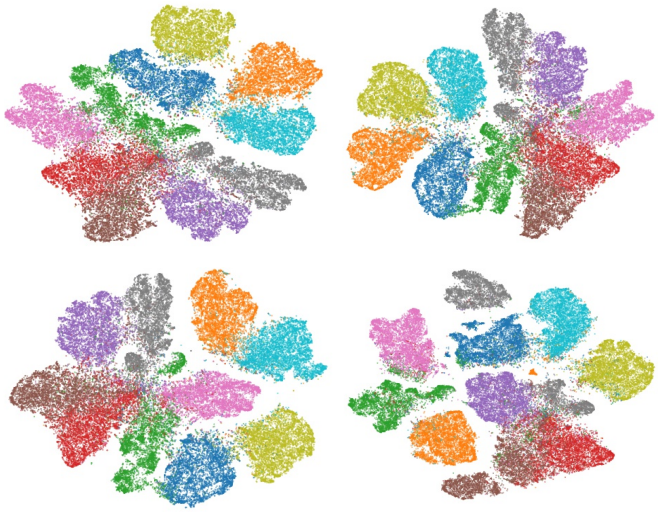

Fig. 6: Comparison of clustering techniuqes on the LeNet [13] codes on the CIFAR-10 dataset. Top-left: SkLearn (3665.44s), Top-right: BH-TSNE (870.71s), Bottom-left: MULTICORE-4(275.46s), Bottom-right: t-SNE-CUDA (Ours, 5.22s).

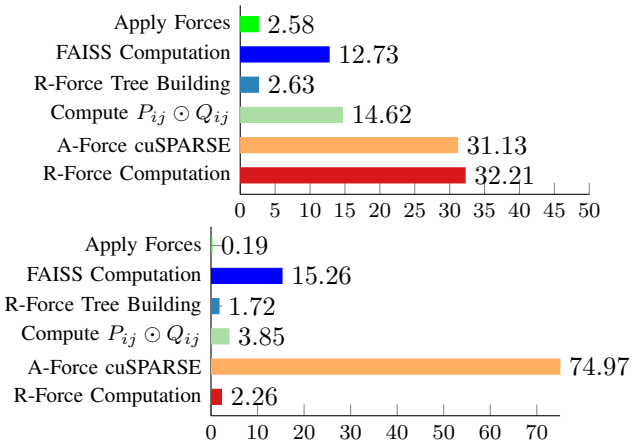

Fig. 7: Percentage of the time spent in each of the top-6 most expensive kernels. The top graph shows these values computed for 500,000 synthetic points, while the bottom graph shows the same computation for 5M points. In both cases the points are sampled from 4 Gaussian distributions of 50 dimensions.

Moreover, the quality of the embeddings produced by t-SNE-CUDA do not differ significantly from the state-of-the-art implementations. Figures 5 and 6 show that t-SNE-CUDA does not compromise on the quality of the clusters while significantly outperforming other state-of-the-art methods in terms of speed.

D. Kernel Performance

Figure 7 gives a general breakdown of the performance of t-SNE-CUDA by percentage of time taken in each part of the algorithm for 500,000 points and 5M synthetic points. There are two phases to t-SNE-CUDA as discussed in Section III, however clearly as the number of points grows larger, the second phase (computing the attractive and repulsive forces) dominates the construction of the nearest neighbors. It is interesting to note that with an increased dataset size, the repulsive force computation time significantly decreased in terms of percentage. The reason for this is that the attractive forces are computed using a sparse matrix multiplication as part of cuBLAS, and the run-time of the Barnes-Hut part of the force computation is dominated by the sparse matrix multiply operation. Future work could improve the sparse matrix multiply operation to further improve performance.

Moreover, cuSPARSE calls dominate the running time for large numbers of points. This is likely due to the fact that the computation of attractive forces breaks into a large sparse matrix dense vector multiplication. Because the sparsity pattern is not well organized (the organization of the sparsity depends on the clustering), it makes it difficult, and rather expensive to compute these values.

Because our kernels are mostly performing integer operations, we achieve a very small amount of the peak theoretical performance of the GPU. Since many of our kernels are memory/offset computations and transforms, we perform very little floating point work, and thus almost all of our kernels

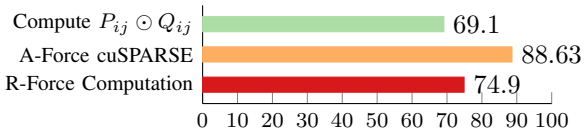

Fig. 8: Percentage of kernel occupancy on the GPU for the top-3 kernels in a run of 500,000 Gaussian symmetric points with four clusters.

achieve $\approx$ 10% of the peak performance of the GPU. The reason that floating point operations are not performed is due to the sparse matrix multiply operation in the attractive force computation kernel. During this sparse multiplication, a large amount of time is spent computing offsets for the different indices, instead of actually computing multiplications. Because this offset-computation performance requires integer arithmetic (which the GPU is not optimized for), we have decreased performance (while performing close to the roof-line for integer computation). Our more heavily floating point focused kernels, such as the integration kernel, can achieve better throughput; For 5M points, we achieve 68.23% of peak performance, close to the roof-line for memory operations. Indeed, our kernels are instead generally memory latency bounded - almost 68.34% of the stalls are caused by memory dependencies. One clear direction for future work is to further examine the memory usage of the project, and to investigate whether better algorithms could be devised to reduce the number of memory stalls.

In general, the kernels that dominate the running time are not troubled by occupancy, and instead are bounded by the sheer number of memory operations (along with the integer arithmetic). Figure 8 shows the occupancy of each of the kernels. By examining the occupancy we can see that while warps may be stalling, we are generally able to have many threads resident at the same time which improves performance. In addition, we find that our average SM utilization is high at above 90% on average for all of the kernels, meaning that we are fully taking advantage of the GPU resources on our device.

V. Exploring Data with t-SNE-CUDA

In this section, we analyze some data visualizations that are made possible with t-SNE-CUDA's improved performance. By leveraging the power to do pixel-level exploration on medium-sized data, and code-level exploration on very large-scale datasets, we can draw interesting conclusions about popular machine learning datasets.

A. Why is CIFAR harder than MNIST?

While it is natural to expect that the CIFAR-10 dataset is much harder than MNIST due to it's dimensionality, this reasoning lies more in intuition than it does in experimentation. The improved efficiency of t-SNE-CUDA allows us to perform pixel-level experimentation in datasets such as CIFAR-10 [11], whereas previously only embedding-level experiments were possible. This allows us to gain additional insight into the

Fig. 9: Raw pixel-space embedding of CIFAR-10 computed using t-SNE-CUDA. Notice that while it does have some local continuity, it does not present clear clustering that MNIST has under the L2 metric.

Fig. 10: Embedding of the 1.2M VGG16 Codes (4096 dim) computed in 523s. We notice that it is relatively more discrete than the ResNet codes shown in Figure 1, perhaps suggesting that the classification space is less continuous under the L2 metric.

reason of CIFAR-10 being a much harder classification problem than MNIST. Since CIFAR-10 is composed of 32x32x3 images, at a pixel level CIFAR has 50K images at 3072 dimensions. Figure 9 shows a t-SNE embedding of the raw pixels CIFAR-10 dataset. We can see immediately from this experiment why classification is much easier on the MNIST dataset. As shown by Figure 5, MNIST has a very clear nearest neighbor structure under the L2 metric in pixel space. In Figure 9, we see that CIFAR does not have the same structure - images that are close in pixel space are likely of many different classes.

While Figure 9 shows that there is clearly some local pixel structure in the dataset, the pixel structure is not as well defined as in the MNIST dataset. Thus, we cannot expect a simple nearest neighbor in the euclidean space to perform well in classification, and we need a non-linear embedding to properly structure the space. Figure 6 shows that our non-linear embeddings provides a better L2 structure for our code, making a nearest neighbor classifier in the code-space more efficient (and validating the power of transforming the data with a neural network).

B. ImageNet

The ImageNet ILSVRC15 dataset [22] is a large-scale image dataset which is particularly popular in computer vision research. ILSVRC15 is composed of 1.2M 224x224x3 full color images. It remains an interesting challenge to explore the ways that different neural networks construct embedding spaces of ILSVRC15. While some previous work has explored codes on the ILSVRC15 validation set [10] - such explorations do not provide a full picture of the embedding space of such a large dataset. Figure 10 shows the embedding of the VGG19 [25] 4096 dimensional codes for the entire ILSVRC15 dataset, while Figure 1 (Page 1) shows the embeddings using ResNet-200 [6].

An interesting aside is that there are many small, tight clusters in the VGG embedding - each corresponding to a different class. In the ResNet embedding, on the other hand, larger clusters are connected by intermediary data points. We find in general that these inter-connected clusters correspond to coarser classifications such as "animals" or "machines." Such more general relationships are not as common in the $L2$ embedding of the VGG codes. These wispy connections suggest that the ResNet embedding space may be more continuous than the VGG embedding space, with points having more inter-class neighbors, while VGG separates classes in a more discrete manner. We can, thus, begin to use the information provided by t-SNE-CUDA to help explore some of the local patterns present in large data/embedding spaces.

C. GloVe

The GLOVE embedding [17] is a natural language dataset with a vocabulary of over 2.2M words, each embedded in 300 dimensional space. GLOVE is a word-similarity embedding trained on 840B tokens found around the internet.

Figure 11 shows a coarse plot of the t-SNE that we computed across the *entire* GLOVE vocabulary. An interactive visualization of this dataset is available at `https://davidmchan.github.io/projects/glove.html`. Our GLOVE embedding was computed in 573.2s. As far as we know, this is the first time that the entire 2.2M dataset has been visualized. We notice that the L2 metric seems to be a questionable choice for comparing GLOVE vectors. While there are nice clusters of textually similar data (such as french words, dates, and times), semantic clusters seem less prevalent in the embedding space, and clusters appear to be

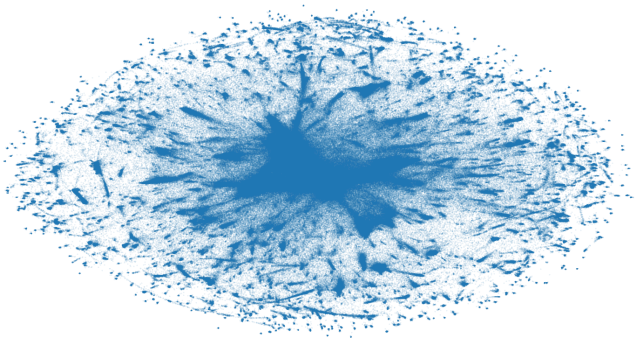

Fig. 11: Embedding of the GLOVE space visualized using t-SNE.

dominated primarily by hamming distance, and not semantic similarity.

VI. Conclusion

In this paper, we have introduced t-SNE-CUDA, a GPU-accelerated implementation of the t-SNE algorithm. We showed that this algorithm can be optimized by using product-quantization to approximate the nearest neighbours of higher dimensional data points and the Barnes-hut method to approximate gradient computation of t-SNE repulsive forces. With these optimizations, we achieved over 50x speedup over state-of-the-art t-SNE implementations and over 650x over the popular SkLearn library. This speedup enables us to explore previously intractable problems - both in the context of vision (with the ImageNet dataset) and NLP (with the GLoVe embeddings). t-SNE-CUDA is publicly available at `https://github.com/CannyLab/tsne-cuda`.

Acknowledgment

The authors would like to thank Dr. James Demmel and Aydın Buluç for their helpful comments and review when writing this paper. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X GPU used for this research. We additionally acknowledge the support of the Berkeley Artificial Intelligence Research (BAIR) Lab.

References

[1] S. Arora, W. Hu, and P. K. Kothari. An analysis of the t-sne algorithm for data visualization. *arXiv preprint arXiv:1803.01768*, 2018.

[2] M. Burtscher and K. Pingali. *An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm*. 2011.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[4] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–647–I–655. JMLR.org, 2014.

[5] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. A. Ranzato, and T. Mikolov. Devise: A deep visual-semantic embedding model. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2121–2129. Curran Associates, Inc., 2013.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] H. Izadinia and P. Garrigues. Viser: Visual self-regularization. 02 2018.

[8] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[9] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[10] A. Karpathy. t-sne visualization of cnn codes. https://cs.stanford.edu/people/karpathy/cnnembed/.

[11] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[12] K. Larsen and J. Nelson. Optimality of the johnson-lindenstrauss lemma. In *58th Annual IEEE Symposium on Foundations of Computer Science*, pages 633–638, 2017.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[15] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[16] T. Mühlbacher, H. Piringer, S. Gratzl, M. Sedlmair, and M. Streit. Opening the black box: Strategies for increased user involvement in existing algorithm implementations. *IEEE transactions on visualization and computer graphics*, 20(12):1643–1652, 2014.

[17] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[18] N. Pezzotti, T. Höllt, J. Van Gemert, B. P. Lelieveldt, E. Eisemann, and A. Vilanova. Deepeyes: Progressive visual analytics for designing deep neural networks. *IEEE transactions on visualization and computer graphics*, 24(1):98–108, 2018.

[19] N. Pezzotti, B. P. Lelieveldt, L. van der Maaten, T. Höllt, E. Eisemann, and A. Vilanova. Approximated and user steerable tsne for progressive visual analytics. *IEEE transactions on visualization and computer graphics*, 23(7):1739–1752, 2017.

[20] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, 2009.

[21] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[23] D. Sacha, M. Sedlmair, L. Zhang, J. A. Lee, J. Peltonen, D. Weiskopf, S. C. North, and D. A. Keim. What you see is what you can change: Human-centered machine learning by interactive visualization. *Neurocomputing*, 268:164–175, 2017.

[24] J. W. Sammon. A nonlinear mapping for data structure analysis. *IEEE Transactions on computers*, 100(5):401–409, 1969.

[25] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] J. B. Tenenbaum, V. De Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[27] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri. Learning spatiotemporal features with 3d convolutional networks. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 4489–4497, Dec 2015.

[28] D. Ulyanov. Multicore-tsne. https://github.com/DmitryUlyanov/Multicore-TSNE, 2016.

[29] L. Van Der Maaten. Accelerating t-sne using tree-based algorithms. *Journal of machine learning research*, 15(1):3221–3245, 2014.

[30] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3357–3364, May 2017.