Improving Barnes-Hut t-SNE Scalability in GPU with Efficient Memory Access Strategies

Bruno Henrique Meyer*, Aurora Trinidad Ramirez Pozo† and Wagner M. Nunan Zola‡
Department of Informatics, Federal University of Paraná, PR, Brazil
Email: *bhmeyer@inf.ufpr.br, †aurora@inf.ufpr.br, ‡wagner@inf.ufpr.br

***Abstract*—The t-Distributed Stochastic Neighbor Embedding (t-SNE) is a widely used technique for dimensionality reduction, however, its application to large datasets is still an issue. In this sense, BH-tSNE was proposed, which is a successful approximation where the Barnes-Hut algorithm is used instead of computing a step of the t-SNE with quadratic computational time complexity. Even so, this improvement still has limitations to process large data volumes (millions of records). Late studies, such as t-SNE-CUDA, have used GPUs to implement highly parallel BH-tSNE. In this research, a new GPU BH-tSNE implementation is proposed using efficient memory access strategies and recent acceleration techniques. Moreover, the embedding of multidimensional data points into three-dimensional space is applied. We examine scalability issues in one of the most expensive steps of GPU BH-tSNE. Our design allows up to 340% faster execution when compared to the t-SNE-CUDA implementation.**

***Index Terms*—Dimensionality reduction, Big data, Visualization of data, t-SNE, BH-tSNE, Barnes-Hut, Parallel algorithms, GPGPU**

I. Introduction

Recently different subareas of Artificial Intelligence and Machine Learning are trying to solve problems containing large datasets which are difficult to process even in modern hardware. Applications such as Deep Convolutional Neural Networks or Deep Reinforcement Learning using these types of big data commonly create intermediate or final results that consists of sets of data points in high dimensional space, which are impossible to interpret and visualize in its original form.

To address this problem, several dimensionality reduction techniques can be used to approximate the structure of these high dimensional spaces by two-dimensional or three-dimensional spaces that can easily be visualized in scatter plots.

The t-Distributed Stochastic Neighbor Embedding (t-SNE) [1] is a dimensionality reduction algorithm that was widely used in machine learning applications due to its efficiency in discovering natural clusters. However, when projecting a large set of data into a lower dimensional space to create an embedded representation from the original data, t-SNE presents quadratic computational time complexity, which is a limiting factor for this algorithm. This problem was differently studied in recent work [2], [3] to investigate the potential of GPU hardware in order to improve the scalability of the algorithm [4]–[7]. The t-SNE-CUDA library[1] [4], [5] utilizes the CUDA language to improve the performance of the approximated Barnes-Hut t-SNE (BH-tSNE) version [2] and FIt-SNE [8] in NVIDIA GPU hardware. The library can accomplish considerable execution speedup, achieving up to 700 times faster execution when compared to other well-known implementations such as the Scikit-Learn library.

The t-SNE-CUDA approach was already compared to different state of the art techniques [5]–[7] but still has potential for improvements. This iterative method runs thousands of steps until convergence. Generally, for new dataset visualizations, it is usual to run the method many times by experimenting with hyperparameters. In this sense, we present an approach that reduces the Barnes-Hut t-SNE computational time, saving considerable time in explorations. Our methods can be used in the same applications as in t-SNE-CUDA. One of these applications is the understanding and inspection of the training process of deep neural networks by visualizing the activation of intermediate or final layers of the network [5].

In this research, we investigate the optimization of the BH-tSNE algorithm implemented in t-SNE-CUDA, adapted to project data into three dimensions, as in Figure 1, while using different strategies proposed in [9] to prevent poor memory access. One of the advantages to project data in three dimensions, rather than two, is the flexibility to represent objects in space with more possibilities for visualization and for configurations, like the disposal of four equidistant points, which is not possible in two dimensions. The major contributions of this work are described as follows:

- We have adapted, implemented and investigated the two-dimensional BH-tSNE projection in t-SNE-CUDA to generate the embedding in three dimensions (3D).
- Regardless of the extra computational effort in embedding to 3D, we have applied acceleration techniques to GPU BH-tSNE algorithm while preserving the quality of the projections.
- It was possible to achieve an speedup of up to 340% in one of the most time-consuming steps of GPU BH-tSNE in 3 million-point datasets, using an Implicit Tree data layout and Simulated Wide-Warp techniques.

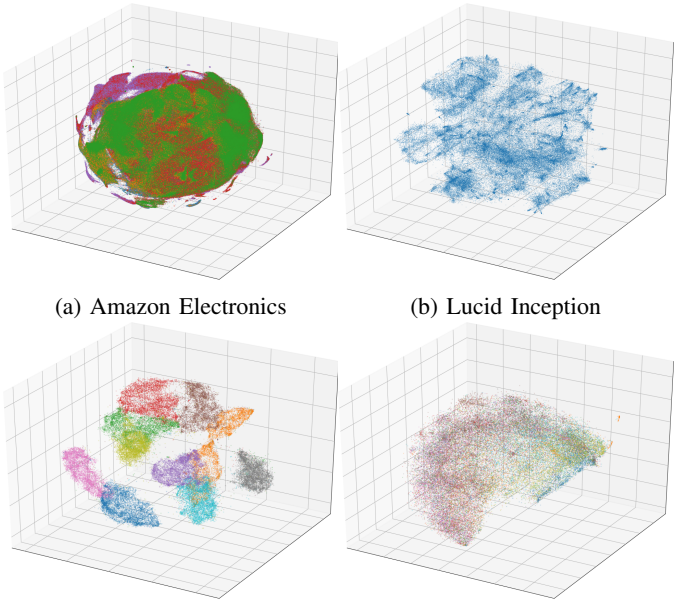
The rest of this work is organized as follows: Section II presents previous research and contributions related to the t-SNE algorithm and its implementation in GPU. Section III

This work was partially supported by Brazilian Education Ministry – CAPES and Brazilian Research Council – CNPq.

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

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

discusses the fundamentals necessary to understand the t-SNE, which will be extended in Section IV to explain the BH-tSNE approximation. We discuss the details of t-SNE-CUDA in Section V and propose our improvements in Section VI. Finally, the methods and experiments used in this study are described in Section VII, with results presented and discussed in Section VIII, followed by conclusions in Section IX.



(a) Amazon Electronics (b) Lucid Inception

(c) MNIST (d) CIFAR

Fig. 1: 3-dimensional embedding generated for different datasets. Each color represent a class for the original instance.

II. Related work

Despite its efficiency in representing high dimensional data points into two or three dimensions, t-SNE [1] suffers from poor scalability when applied to large datasets due to its quadratic time complexity. Since its proposal, many works improve t-SNE efficiency with approximate methods and other methodologies that allow computing the embedding in time $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$.

Currently, Barnes-Hut t-SNE [2], or BH-tSNE, is one of the most well-known implementations of t-SNE whose computational complexity is $\mathcal{O}(N \log N)$. This technique uses Barnes-Hut method [10] to approximate a step in t-SNE called *Repulsive Forces Computation* which is, in the general case, the bottleneck in execution time. The Barnes-Hut algorithm is an approximate method controlled by a parameter θ, which specifies the trade-off between accuracy and time consumption.

Recently, much work has been devoted in proposing methodologies and techniques that efficiently implements the algorithms based on t-SNE with Graphics Processing Units (GPU) [4]–[7]. The utilization of this highly parallel, many-core, hardware enables the use of t-SNE with millions of data. Chan, et al. [4], [5] propose the t-SNE-CUDA, which brings about BH-tSNE in GPU using a modified version of the Barnes-Hut algorithm that was based on the Burtscher and Pingali implementation [11] and allows two-dimensional visualization of the entire ImageNet dataset [12]. Further, t-SNE-CUDA uses FAISS [13] library to compute a K-Nearest Neighbors (KNN) graph in the first step of the algorithm.

Pezzotti, et al. [6] propose an approximation of t-SNE in GPU by rewriting the gradient cost that the original t-SNE specified. The authors discuss and implement a GPU algorithm that has linear time execution complexity and does not depend on CUDA. The experiments discussed by the authors of the GPGPU version presents better preservation of quality in comparison to the implementation in Chan, et al. [4], [5], but cannot surpass t-SNE-CUDA in computational time.

Fu, et al. [7] suggest a technique called AtSNE based on t-SNE, and use GPU to project the data preserving the global structure. The authors discuss the BH-tSNE problems and its limitations like the inefficiency of Barnes-Hut algorithm in GPU. The t-SNE-CUDA [4] implementation was based on the Barnes-Hut algorithm from Burtscher and Pingali [11].

Despite the existence of different works that approximate t-SNE with a linear complexity algorithm like GPGPU t-SNE [5], [6], there is still a need to explore Barnes-Hut t-SNE due to its competitive performance in computational time. Different approaches can be used to improve BH-tSNE, such as the Implicit Tree structure and Simulated Wide-Warp presented by Zola, et al. [9] that mitigates the inefficient GPU memory access existing in common Barnes-Hut implementation.

III. t-SNE

The t-Distributed Stochastic Neighbor Embedding [1] is a technique for dimensionality reduction that aims to preserve the local neighborhood of every projected data point. The preservation of neighborhood can be done by minimizing the difference between two probability distributions that describes the probability of each pair of data points being neighbors in the higher dimension (HD) and in the lower dimension (LD).

The t-SNE method considers a dissimilarity metric d_{ij} between each pair of data points in the HD defined in (1). The dissimilarity is used to compute (2) where p_{ij} is the probability of each point j being neighbor to a point i in a Gaussian distribution, centered in i. The standard deviation σ is calculated with a parameter called Perplexity, specified by the user, that represents the number of effective nearest neighbors around each point.

The t-SNE method differs from Stochastic Neighbor Embedding (SNE) [14] by assuming a t-distribution with one degree of freedom, described in (3), for the probabilities of the neighborhood in the LD instead of a Gaussian distribution as in HD. This assumption leads t-SNE to prevent the "crowding problem" where weak attractions between dissimilar data in HD force the data in LD to become agglutinated [1].

In t-SNE, the Kullback-Leibler divergence is used to measure the difference between the distribution probability of the neighborhood of a specific data point in HD and its correspondent in LD, noted as P_i and Q_i in (4). This measure

can be used to estimate a cost function that describes how well the projected data point in LD represent the data point in the HD using (5). Analytically, as demonstrated by Van der Maaten and Hinton [1], it is possible to estimate the error contribution of each data point relative to the cost function. This contribution is represented in (6), and can be used to "move" the points in the LD in order to reduce the Kullback-Leibler divergence.

Stochastic Gradient Descent (SGD) is a widely used algorithm for search for minimum and local minima in convex and non-convex differentiable functions [15]. Equation (6) can be used to minimize the cost function in (5) by iteratively updating the points in LD with SGD, which is the technique applied in the t-SNE Algorithm 1 [1].

$$d_{ij}^2 = \frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma_i^2} \tag{1}$$

$$p_{ij} = \frac{\exp(-d_{ij}^2)}{\sum_{k \neq i} \exp(-d_{ik}^2)} \tag{2}$$

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}} \tag{3}$$

$$KL\left(P_i \| Q_i\right) = \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{4}$$

$$C = \sum_i KL\left(P_i \| Q_i\right) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{5}$$

$$\frac{\delta C}{\delta y_i} = 4 \sum_j \left(p_{ij} - q_{ij}\right)\left(y_i - y_j\right)\left(1 + \|y_i - y_j\|^2\right)^{-1} \tag{6}$$

The algorithm that implements t-SNE accepts the original set of points S with high dimensionality as parameter and other arguments that control the convergence of the algorithm. The step size (or learning rate) η is used to multiply the "movement" of each point in the lower dimension using its impact in (6). Also, the momentum α_t at each iteration t is used to prevent poor local minima in the optimization using the movement from the previous iteration.

Initially, the algorithm computes the p_{ij} of each pair of points using the Perplexity specified in parameters. This step can be executed only once at the beginning of the algorithm. Then, an initial random projection is created, usually generated using a Gaussian distribution, and the SGD is executed until the KL Divergence achieves a small value or a maximum number of steps is reached.

A. t-SNE projection quality

Since the t-SNE aims to reduce the dimensionality of a higher dimensional space by preserving the local neighborhood of each original point, different metrics can be used to compute the projection quality. One of these metrics is the rescaled average K-ary neighborhood preservation [16] presented in (8), where the average represents the intersection

Algorithm 1 t-SNE

Input: HD points $S \in \mathbb{R}^D$, Step size η, Momentum α, LD size D', Maximum iterations max_ite, Perplexity $Perpl$
Output: LD points $S' \in \mathbb{R}^{D'}$

```
Calculate p_ij for each pair of points in S using perplexity Perpl (Equation 2)
Y_0 ← initial_random_projection( )
Y_1 ← Y_0
t ← 2
repeat
    Calculate q_ij for each pair of points in S' (Equation 3)
    Calculate the gradient ∂C/∂y_i for each point y_i ∈ Y_t (Equation 6)
    Y_{t+1} ← Y_t = Y_t + η δC/δY + α_t (Y_t − Y_{t−1})
    t ← t + 1
until Convergence or t > max_ite
return Y_t
```

of the K neighbors ν_i^K of each point of index i in the HD space and n_i^K the K neighborhood in LD space.

This metric is represented in (7), where the total of points N and the neighborhood size K are used to normalize the metric, which will be rescaled in (8). Therefore, the $R_{\mathrm{NX}}(K)$ metric grows with the preservation of the K closest points by the projection of HD points into a LD space. Note that this metric has value equal to 1 when the neighbors of every points was successfully preserved.

$$Q_{NX}(K) = \frac{1}{KN} \sum_{i=1}^{N} \left|\nu_i^K \cap n_i^K\right| \tag{7}$$

$$R_{\mathrm{NX}}(K) = \frac{(N-1)Q_{\mathrm{NX}}(K) - K}{N - 1 - K} \tag{8}$$

IV. Barnes-Hut t-SNE

One major contribution for t-SNE implementations was the Barnes-Hut t-SNE (BH-tSNE) proposed by Van der Maaten [2]. The work proposes an approximate algorithm that implements t-SNE with complexity $\mathcal{O}(N \log N)$ by using a KNN algorithm to approximate the computation of p_{ij} and Barnes-Hut algorithm to approximate q_{ij}. This is done by rewriting (6) as a sum of two terms, F_{attr} (10) and F_{rep} (11), representing forces that must be respectively applied to approximate points that are distant in LD and close in HD, and to separate points that are close in LD and distant in HD, as in (12).

In the original proposal, the KNN is computed using a vantage point tree in $\mathcal{O}(uN \log N)$ time, where u represents the Perplexity. KNN produces the $\lfloor 3u \rfloor$ closest neighbors for each point in HD, considering $p_{ij} = 0$ for every pair of points that are not neighbors. This assumption does not compromise the results of t-SNE because distant points, and consequently not neighbors, already have similarities p_{ij} close to 0.

A. Barnes-Hut Algorithm

The Barnes-Hut algorithm [10] is an approximate method for N-Body simulation, where N different bodies (particles)

constantly interact applying forces between each other. The idea is to reduce the quadratic nature of the common algorithm by approximating weak interactions between distant clusters using an m-ary Tree to create different partitions of bodies. The 3-dimensional three is an Octree where leaf nodes are real bodies, and internal nodes summarize octant (cells) information such as centroid position, the cell radius, total contained mass, and other relevant information on each sub-tree.

Once the tree is built, it is traversed in preorder while computing forces applied to each body. Algorithm 2 shows the traversal and forces calculations steps, where an approximation can be made by checking if the descendant nodes of each centroid are too distant from the reference body. This verification is done by using (13) where $\theta \in [0, 1]$ is a threshold that controls the approximation. Note that the relationship defined in (13) will tend to be satisfied when the distance between a query point y_i and a centroid y_{cell} increases and the radius r_{cell}, related to y_{cell}, decreases. When the equation is satisfied, the algorithm estimates the interaction forces between y_i and all y_{cell} descendants by using y_{cell} center of mass.

$$Z = \sum_{k \neq l} \left(1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2\right)^{-1} \tag{9}$$

$$F_{attr} = \sum_{j \neq i} p_{ij} q_{ij} Z \left(\mathbf{y}_i - \mathbf{y}_j\right) \tag{10}$$

$$F_{rep} = -\sum_{j \neq i} q_{ij}^2 Z \left(\mathbf{y}_i - \mathbf{y}_j\right) \tag{11}$$

$$\frac{\partial C}{\partial \mathbf{y}_i} = 4 \left(F_{attr} + F_{rep}\right) \tag{12}$$

$$\frac{r_{cell}}{\|\mathbf{y}_i - \mathbf{y}_{cell}\|^2} < \theta \tag{13}$$

This approximation gives the algorithm a computational time complexity $\mathcal{O}(N \log N)$, controlled by θ. Note that when $\theta = 0$ the approximation gives the exact quadratic algorithm, justifying the use of high values for θ (usually 0.5) to reduce the computational time of the algorithm considering a trade-off between accuracy and execution time.

B. t-SNE repulsive forces

The repulsive forces applied to each point in LD represented in (11) can be interpreted as an N-Body simulation problem. Using the Barnes-Hut Algorithm to approximate the repulsive forces by manipulating (3) to compute the interaction between different nodes, the repulsive forces computed in each step of the stochastic optimization can be made in $\mathcal{O}(N \log N)$ computational time, and consequently determining the complexity of t-SNE algorithm [2]. Note that the Z term presented in (9), used in the repulsive and attractive forces, can be computed only once during the tree traversal by ignoring distant points.

V. t-SNE-CUDA

Chan, et al. [4] proposed t-SNE-CUDA, and a subsequent improved version [5] which implements BH-tSNE algorithm using GPU and CUDA primitives. FIt-SNE [3] was also implemented in GPU. In t-SNE-CUDA, the FAISS library [13] is used to compute an approximate KNN of HD points with GPU parallelism in linear computational time complexity [5].

Algorithm 2 Recursive Barnes-Hut Tree traversal

Input: Body y_i, m-aryTree, Current node y_j, Threshold θ
Output: Forces vector with average forces applied into y_i

```
1:  C ← children(y_j)
2:  Forces ← 0⃗
3:  for all c ∈ C where y_i ≠ c do
4:      r_cell ← radius(cell(c))
5:      y_cell ← centroid(cell(c))
6:      if Equation 13 is not satisfied then
7:          Forces ← Forces + Algorithm 2(y_i, m-aryTree, c, θ)
            // Recursive call
8:      else
9:          // Approximates the interaction with distant cells
10:         Forces ← Forces + interaction(y_cell, y_i)
11:     end if
12: end for
13: return Forces
```

The original implementation[2] allows t-SNE-CUDA to create the projection only to a two-dimensional space. The computation of attractive forces at each SGD step on this work uses cuBLAS library to apply a sparse matrix multiplication, which was improved in [5] using GPU atomic operations.

As described in the original article, the Barnes-Hut steps are mainly composed by:

- Finding the bounding box of all points: The result of this step is used to construct initial cells of the Quadtree;
- Construction of the Quadtree;
- Nodes summarization: Computation of the radius, centroid and the number of points of each cell;
- Sorting of points in spatial order (Morton Order);
- Computation of the repulsive forces: Traverses the tree while evaluating forces for each body (Algorithm 2).

In current NVIDIA GPU architectures, there is a concept called *warp* that represents a set of 32 cores executing the same SIMD (*Single Instruction Multiple Data*) instruction at a given time. When cores of the same warp try to follow different paths in the code, the GPU must necessarily serialize the execution of some cores. This partially wastes the potential parallel advantage of GPU SIMD hardware, leading to a problem called thread divergence. Sorting points in Morton order is an optional step, but it improves the chances that threads with close indices in the GPU agree in expanding the same cells (line 6 of Algorithm 2), and consequently reduces divergence.

[2]https://github.com/CannyLab/tsne-cuda/tree/bh_tsne.

VI. Implicit Tree and Simulated Wide-Warp

The t-SNE-CUDA Barnes-Hut code is based on the work of Burtscher and Pingali [11] (BP-BH), with specific forces calculations modified for the t-SNE method and with dimensionality of N-Body simulation simplified to 2 dimensions. These implementations are iterative, instead of recursive as Algorithm 2, and are implemented with a stack data structure in the traversal. The tree representation is *Sparse* and uses an array of integer pointers to child nodes in each non-leaf node.

Zola, et al. [9] presented a GPU implementation of the Barnes-Hut algorithm that surpasses BP-BH in computational time by using a different tree representation called *Implicit Tree*. Moreover, they proposed *Simulated Wide-Warp* (SWW), a technique that creates virtual threads simulating warps with more threads per core than what is physically supported by the GPU. These modifications allow BH algorithm to present better memory access patterns and to acquire more accurate results with considerable speedup.

A. Implicit Tree

In the Implicit Tree representation, leaf nodes and internal tree nodes are represented in an unique array, where internal nodes contain a pointer to the next internal node, assuming a pre-order traversal. Figure 2 illustrates the difference between Sparse and Implicit representations. The Implicit representation needs to store only one pointer in each non-leaf node. This is an advantage that reduces the needed memory in comparison to the Sparse representation, where even null nodes need to be represented. While saving space allowing larger trees and posing less bandwidth demand on the memory system, another benefit of the implicit representation is noted when traversing the tree in the BH algorithm: unlike the Sparse representation, there is no need to use a stack per thread when traversing the tree since the *skip-link* pointers in each internal node are the only information needed to walk through the tree. This allows the algorithm to access GPU memory more consistently, reducing register use, global memory accesses and execution time.

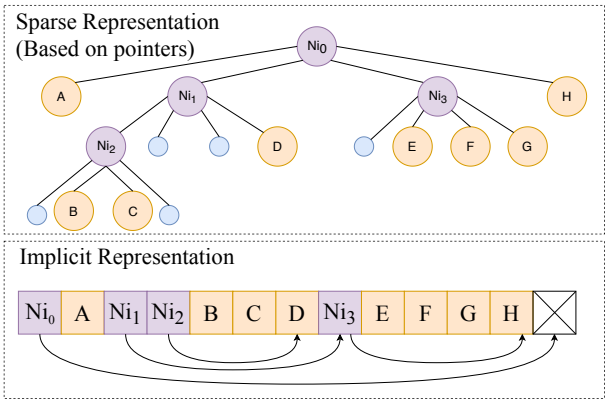


Fig. 2: Types of tree representation in Barnes-Hut algorithm. The example illustrates a Quadtree with 8 leaves (in orange). Internal nodes are represented in purple and null nodes in blue.

B. Simulated Wide-Warp (SWW)

Modern GPU architectures allow a large number of threads but the actual number of active threads at a given time is proportional to the number of cores. The Simulated Wide-Warp technique consists in simulating extra virtual threads by exploring the large number of GPU registers (current up to 65536). This is done in algorithm by carefully coding specific loop unroll patterns that are not automatically captured by current compilers.

The simulation of virtual threads is done inside each physical thread. Virtual threads allow sharing thread local variables without additional real memory access. Basically, SWW is used to optimize forces calculation with minor changes to the implicit tree traversal steps in algorithm BH. The programmer must be careful to create specific variables needed by each virtual thread and not surpass the GPU register limit, otherwise leading to a problem called register spill where the extra local variables will be placed in global memory.

In SWW, the effective number of simulated threads corresponds to the amount of physical warp threads multiplied by a constant number (*WarpWidth*) of virtual threads implemented per physical thread. To lower thread divergence, the tree traversal in SWW-tSNE must ensure that all threads in a warp execute the same instruction. This can be ensured if at least one thread chooses to expand a node. In this case, all threads in the simulated warp must perform the same expansion, even if (13) is not satisfied. With this guarantee, the algorithm can perform faster and return more accurate results. This is also ensured in t-SNE-CUDA but its efficiency in SWW-tSNE grows with the width of the virtual warp.

Since the coding of repulsive forces in t-SNE differs from the computation of gravitational forces in the traditional BH, the number of registers used per thread also tends to be different. As such, the impact of SWW to t-SNE algorithm needs to be verified to access potential speedup and quality of algorithm convergence for different values of WarpWidth when processing the standard datasets with diverse sizes.

VII. Methods

The original implementation of t-SNE-CUDA was modified to project the input data points of the algorithm into a tridimensional space, which we will call “Original version”. We produced a new version of GPU BH-tSNE that performs the BH step using implicit trees and conducts the tree walking and forces calculations phase using SWW techniques, wich we will call *SWW-tSNE*. Then, different SWW-tSNE versions were created, increasing WarpWidth in each version, using the maximum number of threads per block and, consequently, the maximum number of registers available in the GPU, achieving up to WarpWidth=4 before the register spill problem occurs. It is essential to mention that in the experiments of this work we will consider only the optimization of the Tree Building and Traversal present in the Barnes-Hut algorithm. Also, note that the WarpWidth=1 version is equivalent to the Original, to the extent that it does not simulate larger than physical warps and the only difference is the Tree structure used by the algorithm.

The computation of the attractive forces in the Original version is still using a deprecated implementation with cuBLAS [4] that was improved in recently works [5] using GPU atomic operations, which was also implemented in our algorithms.

A. Environment

All experiments reported in this work were performed in a 3.20GHz i5-4460 processor with 4 CPU cores, 8GB of processor RAM, and GPU NVIDIA GeForce RTX 2070 with 8GB of GPU RAM, running CUDA 10.1 tools. We used GNU GCC 4.8, and G++ 4.8 to compile each version of t-SNE.

B. Datasets

In order to observe the behavior of proposed modifications in t-SNE-CUDA, datasets of different sizes and dimensions were used, presented in Table I. The MNIST [17] and CIFAR-10 [18] datasets are widely used in supervised machine learning research, which consists of a set of 60000 and 50000 records that represent raw image data, both with ten different classes.

The Lucid Inception dataset represents the data extracted from the Lucid library[3], which provides the activation of intermediate layers of Convolution Neural Networks (CNN) models, by collecting the activation of 100000 images from the ImageNet dataset in the Google Inception V1 CNN architecture [19].

A similar methodology proposed by Fu, et al. [7] was used to create the Amazon Electronics dataset. The FastText library [20] was used to create a 100-dimensional text embedding of the 1689188 text reviews of electronic products from Amazon web store[4] [21] in which every review also contains an integer value between 1 and 5 that represents the overall rate of the review.

The GoogleNews300 dataset[5] consist in the 100 dimensional word embedding of 3 million words created with the Word2Vec [22] model using the Google News text dataset.

TABLE I: Dataset sizes used in experiments.

Dataset name	Total of points	Total of dimensions
CIFAR	50000	3072
MNIST	60000	784
Lucid Inception	100000	128
Amazon Electronics	1689188	100
GoogleNews300	3000000	300

C. Evaluation

Each version of t-SNE-CUDA was compared using the datasets described before. The t-SNE was executed for 1000 iterations using the default parameters of t-SNE-CUDA, where the KNN was limited to compute the 32 nearest neighbors of each point due to the limited memory size of GPU. For each dataset, the versions implemented where executed ten times in order to achieve more accurate analysis and three times in the GoogleNews300 dataset due to its size and our limited resources.

We split the algorithms in five main steps and the computational time of every step was measured. The algorithms were split as follows:

- KNN: Computation of the nearest neighbor of each data point in the high dimensional space. The result is used to precompute the p_{ij} values;
- Attractive Forces: Computation of (10) for each data point using the p_{ij} values;
- Tree Building: Step where all operations necessary to create the Octree are executed.
- Tree Conversion: Necessary step to convert the Sparse tree into an Implicit tree. This step is not needed in the Original version and can be removed in future SWW-tSNE implementations by using a direct massively parallel algorithm to construct the Implicit Tree, in this case, the construction of the sparse tree will also be unnecessary in the SWW version, further saving execution time.
- Tree Traversal: The computation of repulsive forces described in (11) using the Barnes-Hut algorithm by executing a Traversal in the corresponding Octree for each data point.

VIII. Results and Discussion

The usage of Implicit Tree structure and Simulated Wide-Warp strategies have demonstrated to be well suited to improve the scalability of t-SNE-CUDA without the necessity of rewriting or approximating the computations of t-SNE.

Table II shows the speedup in the Tree Traversal step in which it was possible to achieve an execution up to 3 times faster than the Original version of t-SNE-CUDA to perform the projection in three dimensions. The Simulated Wide-Warp technique was not wholly successful in the smaller datasets compared to the million size datasets, because when the Warp-Width parameter grows, the number of threads to process the computation of each point also increases. When the number of threads surpasses the real number of points to process, several threads will be idle. Therefore, the technique will not provide improvement of performance and possibly creating a computation overhead without benefits. This problem can be prevented by searching the ideal WarpWidth before the execution of t-SNE by using the information of the current GPU architecture and the number of points to be processed.

Considering the most expensive steps of t-SNE-CUDA observed in the experiments, the KNN is one of the critical steps once it depends on the dimensionality of the dataset. This step can be the bottleneck of the algorithm in some cases.

Fortunately, this problem can be addressed by using multiple GPUs [5] as described by the author of the technique or by the use of other algorithms to compute the approximated KNN rather than the FAISS library.

Figure 1 illustrates the result obtained from the execution of t-SNE-CUDA on the Amazon Electronics dataset, and Figure 3 shows the time necessary for the execution of the

[3]https://github.com/tensorflow/lucid

[4]http://jmcauley.ucsd.edu/data/amazon/

[5]https://code.google.com/archive/p/word2vec/

principal steps of the algorithm. In scenarios where Barnes-Hut algorithm is improved, the other steps (different from the Tree Traversal and forces calculations) may become the bottleneck of execution time.

In order to overcome this problem, a similar effort done in this work can be made in these steps by exploring the GPU parallelism. Further, other approximations strategies that do not harm the quality of the projection result can be explored.

The proposed modifications in SWW-tSNE were implemented in an environment wherein the parameter WarpWidth of the Simulated Wide-Warp technique was limited to create a maximum of four virtual threads for each physical thread before all registers available in the GPU were utilized. One of the most limiting factors to explore values greater than 4 is the use of local variables that grows linearly with the WarpWidth and dimensions of the lower-dimensional space. It is expected that when the same methodology is used to implement a 2-dimensional projection algorithm in SWW-tSNE, a higher number of virtual threads could be created, which would allow even greater speedups than the achieved for three dimensions.

In order to inspect the projection quality and convergence of each implementation, we analyze the results of each dataset except GoogleNews300 due to its large size. The convergence was interpreted as the average magnitude of forces applied to each point during the iterations of the algorithm, present in the original t-SNE-CUDA library, which is called "gradient norm" and the quality as the $R_{\mathrm{NX}}(32)$ described in (8).

Table III contains the average and standard deviation of the $R_{\mathrm{NX}}(32)$ in each dataset with different WarpWidth implementations, in which WarpWidth=3 seems to create better results in smaller datasets and WarpWidth=1 in the Amazon Electronics dataset. However, the Kruskal-Wallis H test [23] with a significance level equal to 0.05 indicates that there is not any statistical difference between different WarpWidth implementations in each dataset.

Also, this difference does not seem to impact the convergence of the algorithm significantly, which is represented in Figure 4 that contains the result of one execution of t-SNE in MNIST dataset for each implementation. The gradient norm convergence is shown in Figure 4e. In Figure 4, it is possible to note that the number of iterations is more relevant than the difference between each version of SWW-tSNE implemented in this study.

These results are not expected since the Barnes-Hut must be more precise with higher values of WarpWidth [9]. This characteristic can be explained by the fact that soft adjustments in the approximation of the repulsive forces do not necessarily impact positively in the algorithm convergence. Since the t-SNE try to optimize a non-convex function and it is more susceptible to other parameters like the total number of iterations, learning rate, and projection initialization. Future works can explore these characteristics and create better approximations of Barnes-Hut t-SNE in GPU in order to improve its scalability without harming the quality of projection or the convergence of the algorithm.

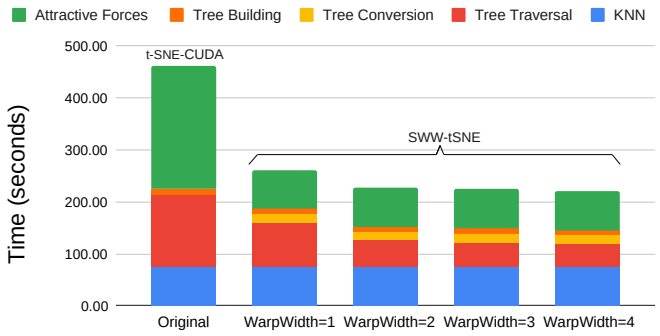


Fig. 3: Execution time breakdown of different GPU t-SNE versions applied to Amazon Electronics dataset. *Original* represents the standard t-SNE-CUDA and the other versions represent our implementations using Implicit Tree data structure and Simulated Wide-Warp with different WarpWidth sizes.

TABLE II: Average speedup achieved and (standard deviation) in the Barnes-Hut Tree Traversal in the different versions of t-SNE-CUDA. WarpWidth is abbreviated as WW.

Dataset name	Original	WW1	WW2	WW3	WW4
GoogleNews300	1	1.713 (0.001)	2.929 (0.003)	3.261 (0.001)	**3.404** (0.015)
Amazon Electronics	1	1.618 (0.004)	2.703 (0.007)	2.949 (0.014)	**3.052** (0.011)
Lucid Inception	1	1.042 (0.013)	1.245 (0.017)	**1.535** (0.021)	1.217 (0.020)
MNIST	1	0.973 (0.006)	**1.379** (0.023)	1.019 (0.018)	0.803 (0.017)
CIFAR	1	0.965 (0.009)	**1.352** (0.019)	0.979 (0.013)	0.768 (0.013)

IX. Conclusions and Future Work

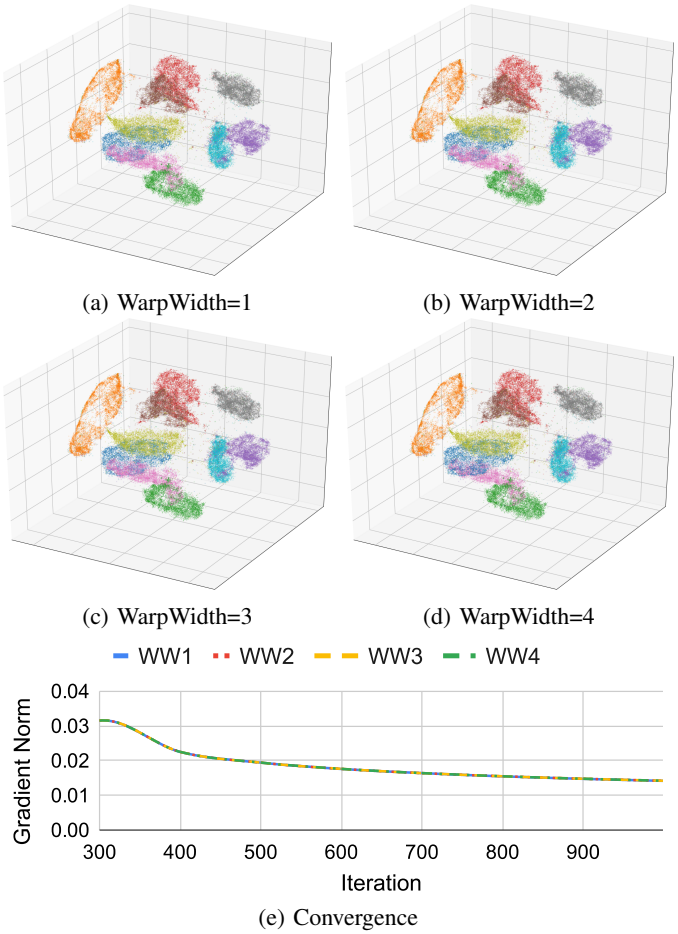
Despite t-SNE-CUDA efficiency and competitiveness among state-of-the-art t-SNE implementations in GPUs, to the best of our knowledge, there is no previous work verifying the impact of Implicit Tree representation and SWW techniques in GPU t-SNE.

In this research, we have investigated the advantages of these techniques considering the potential speedup in the Barnes-Hut step of t-SNE, the quality of the algorithm convergence and scalability to large datasets.

One of the most computationally expensive steps of t-SNE-CUDA is the calculation of repulsive forces while traversing

TABLE III: Average $R_{\mathrm{NX}}(32)$ and (std. deviation) achieved.

Dataset name	WW1	WW2	WW3	WW4
Amazon Electronics	**0.1325429** (0.0013978)	0.1325359 (0.0013952)	0.1325225 (0.0013954)	0.1325358 (0.0013911)
Lucid Inception	0.1969604 (0.0006278)	0.1969638 (0.0006124)	**0.1970092** (0.0006483)	0.1969707 (0.0006336)
MNIST	0.3578996 (0.0010927)	0.3579124 (0.0010830)	**0.3579271** (0.0010848)	0.3579073 (0.0011002)
CIFAR	0.1326866 (0.0009716)	0.1327137 (0.0009598)	**0.1327733** (0.0009317)	0.1327453 (0.0009426)



(a) WarpWidth=1 (b) WarpWidth=2

(c) WarpWidth=3 (d) WarpWidth=4

(e) Convergence

Fig. 4: Result of the 3-dimensional embedding and convergence generated for MNIST datasets using different versions and fixed random seed.

the tree in the modified Barnes-Hut phase, which is a memory bound step. Our current work further contributes to mitigate memory accesses and reduce execution time. Our experiments demonstrate that strategies such as the usage of the Implicit Trees and Simulated Wide-Warp can speedup the tree traversal and repulsive forces calculations step in up to 340% in 3 million point datasets. We would like to further study the effectiveness of SWW applied to other steps of t-SNE-CUDA.

Notwithstanding the efficiency observed in the experiments, only three-dimensional projections were implemented in this work. However, we are confident that the two-dimensional embedding can take greater advantage of the Simulated Wide-Warp technique and achieve better speedups. This is due to the fact that the SWW calculations in the 2-dimensional tree utilizes fewer registers and demands less memory bandwidth due to smaller trees. Further, several aspects can be explored to improve the scalability of Barnes-Hut t-SNE in GPU without compromising the quality of the projection, like the fact that Simulated Wide-Warp enables the communication of different virtual threads in GPU without a significant overhead, which can be explored in future investigations.

References

[1] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[2] L. Van Der Maaten, "Accelerating t-SNE using tree-based algorithms," *Journal of Machine Learning Research*, vol. 15, pp. 3221–3245, 2014.

[3] G. C. Linderman, M. Rachh, J. G. Hoskins, S. Steinerberger, and Y. Kluger, "Efficient algorithms for t-distributed stochastic neighborhood embedding," *ArXiv*, vol. abs/1712.09005, 2017.

[4] D. M. Chan, R. Rao, F. Huang, and J. F. Canny, "t-SNE-CUDA: GPU-Accelerated t-SNE and its applications to modern data," *Proceedings - 2018 30th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2018*, pp. 330–338, 2018.

[5] ——, "GPU accelerated t-distributed stochastic neighbor embedding," *Journal of Parallel and Distributed Computing*, vol. 131, pp. 1–13, 2019.

[6] N. Pezzotti, J. Thijssen, A. Mordvintsev, H. Thomas, B. V. Lew, B. P. F. Lelieveldt, E. Eisemann, and A. Vilanova, "GPGPU linear complexity t-SNE optimization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2626, no. c, 2019.

[7] C. Fu, Y. Zhang, D. Cai, and X. Ren, "AtSNE: Efficient and robust visualization on GPU through hierarchical optimization," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 25, pp. 176–186, 2019.

[8] X. Shen, X. Zhu, X. Jiang, T. He, and X. Hu, "Visualization of disease relationships by multiple maps t-SNE regularization based on Nesterov accelerated gradient," *Proceedings - 2017 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2017*, vol. 2017-Janua, no. 2, pp. 604–607, 2017.

[9] W. M. N. Zola, L. C. E. Bona, and F. Silva, "Fast GPU parallel n-body tree traversal with Simulated Wide-Warp," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2015-April, pp. 718–725, 2014.

[10] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[11] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU computing Gems Emerald edition*. Elsevier, 2011, pp. 75–92.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

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

[14] G. Hinton and S. Roweis, "Stochastic neighbor embedding," *Advances in Neural Information Processing Systems*, 2003.

[15] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[16] J. A. Lee, D. H. Peluffo-Ordóñez, and M. Verleysen, "Multi-scale similarities in stochastic neighbour embedding: Reducing dimensionality while preserving both local and global structure," *Neurocomputing*, vol. 169, pp. 246–261, 2015.

[17] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[18] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[20] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[21] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel, "Image-based recommendations on styles and substitutes," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2015, pp. 43–52.

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[23] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.