Karlsruhe Institute of Technology

Institute for Automation and Applied Informatics

Fast Visualization of High-Dimensional Data via Parallelized UMAP on GPUs

Master Thesis of

Peter Eisenmann

At the Department of Informatics
Institute for Automation and Applied Informatics

Reviewer 1: Prof. Dr. Veit Hagenmeyer
Reviewer 2: Prof. Dr. Achim Streit
Advisor: Andreas Bartschat

Time Period: 1st November 2018 – 31th May 2019

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 30.05.2019

Abstract

Visualization is important for humans to perceive, analyze and understand data. Exploratory Data Analysis is essential in fields such as neural networks, deep-learning and data science in general. Unfortunately data in these fields commonly has hundreds or thousands of dimensions, making classic visualization techniques unsuitable. In recent years the dimensionality reduction algorithm t-SNE gained popularity. It preserves local structures, allowing for meaningful visualizations of complex data. UMAP is a new algorithm, that challenges t-SNE as the status quo It provides better performance in both, execution time and quality of graphical results. This thesis explores the functionality of UMAP and tries to improve its performance further, by parallelizing it to execute on graphics cards, GPUs. Profiling the GPU version, GPUMAP, shows speedups of up to sixty times on current hardware.

Zusammenfassung

Visualisierungen sind wichtig für Menschen um Daten zu betrachten, zu analysieren und zu verstehen. Explorative Datenanalyse ist essentiell in Bereichen wie Deep-Learning, Neuronale Netze und Data Science allgemein. Allerdings haben Daten in diesen Bereichen oft hunderte oder tausende von Dimensionen, was es klassischen Visualisierungstechniken unmöglich macht, sie darzustellen. In den letzten Jahren gewann t-SNE als eine Methode zur Dimensionsreduktion an Beliebtheit. t-SNE erhält lokale Strukturen, was eine aussagekräftige Visualisierung von komplexen Daten ermöglicht. UMAP ist ein aktuell veröffentlichter Algorithmus, der t-SNE als den Status Quo infrage stellt. Er bietet bessere grafische Ergebnisse bei zugleich schnellerer Ausführung. Diese Arbeit untersucht die Funktionalität von UMAP und versucht durch eine Parallelisierung des Algorithmuses auf Grafikkarten (GPUs), dessen Leistung noch weiter zu verbessern. Messungen der GPU-Version GPUMAP zeigen bis zu sechzigfache Beschleuningungen auf aktueller Hardware.

Contents

1 Introduction **8**
1.1 Background and Motivation . . . 8
1.2 Related Works . . . 9
1.2.1 Linear tSNE Optimization for the Web . . . 9
1.2.2 t-SNE-CUDA: GPU-Accelerated t-SNE and its Applications to Modern Data . . . 11
1.2.3 Comparison of TensorFlow t-SNE and t-SNE-CUDA . . . 13
1.3 Aim of the Thesis . . . 13
1.3.1 Thesis Outline . . . 14

2 Preliminaries **15**
2.1 Clustering Algorithms . . . 15
2.1.1 K-Means . . . 15
2.1.2 DBSCAN . . . 18
2.2 Dimensionality Reduction Algorithms . . . 21
2.2.1 PCA . . . 21
2.2.2 t-SNE . . . 23
2.2.3 UMAP . . . 27
2.2.4 Visual Results . . . 30
2.3 Algorithm Complexity Overview . . . 31

3 Methods **32**
3.1 Profiling UMAP . . . 32
3.1.1 Identifying Subtasks of UMAP . . . 32
3.1.2 Influence of Data Set Size and Dimensionality . . . 34
3.1.3 Variation of Execution Time . . . 37
3.2 Parallelizing Subtasks of UMAP . . . 37
3.2.1 Parallelization of KNN Search . . . 37
3.2.2 Parallelization of Distance Normalization . . . 38
3.2.3 Parallelization of Embedding Initialization . . . 39
3.2.4 Parallelization of Embedding Optimization . . . 39
3.3 Coverage of Subtasks . . . 42

4 Implementation **43**
4.1 Used Technologies . . . 43

4.2 Implementation Details 44
4.2.1 Fallback Routines 44
4.2.2 Subtask `nearest_neighbors` 44
4.2.3 Subtask `fuzzy_simplicial_set` 45
4.2.4 Subtask `simplicial_set_embedding` 46
4.2.5 Subtask `optimize_layout` 46

5 Results **50**
5.1 Performance 50
5.2 Visualization Quality 52

6 Discussion **55**
6.1 Advantages of GPUMAP 55
6.2 Drawbacks of GPUMAP 55

7 Conclusion **57**
7.1 Summary 57
7.2 Future Works 57
7.3 Closing Words 58

Bibliography **59**

List of Tables

1.1 Performance comparison of different t-SNE implementations. 13

2.1 Comparison of Time and Space Complexities for presented Algorithms. . 31

3.1 Profiling of UMAP on various data sets. 34

5.1 Profiling of GPUMAP on various data sets. 50
5.2 Comparison of GPUMAP and t-SNE-CUDA on the CIFAR, MNIST and GloVe data sets. 52

List of Figures

1.1 Visualization of an MNIST subset by Pezzotti et al.'s t-SNE variation. . . 11

2.1 Examples of combining clustering algorithms with visualization techniques. 16
2.2 Convergence of the K-Means algorithm. 18
2.3 Example of non-convex clusters found by DBSCAN. 19
2.4 Visualizations of the COIL-20 data set. 24
2.5 Tree structure used by the Barnes-Hut algorithm. 25
2.6 Exemplary graph created by UMAP for data lying on a sine wave manifold. 28
2.7 Overcoming the curse of dimensionality with locally normalized distances. 29
2.8 Visualizations of data sets by PCA, t-SNE and UMAP. 31

3.1 Profiling of UMAP on a 500.000 points subset of GoogleNews. 33
3.2 Profiling of UMAP on varying sizes and dimensions of CIFAR. 35
3.3 Time spent by individual subtasks processing reduced CIFAR data sets. . 36
3.4 Comparison of UMAP subtasks on reduced CIFAR data sets. 36
3.5 Standard deviation of UMAP execution times on reduced CIFAR data sets. 37
3.6 Time spent by `fuzzy_simplicial_set` when processing 500.000 points of the GoogleNews data set. 38
3.7 Time spent by `simplicial_set_embedding` when processing a 500.000 points subset of GoogleNews. 39
3.8 Comparison of UMAP subtasks on reduced GoogleNews data sets. 40
3.9 Conceptual depiction of edge pre-filtering. 41
3.10 Parallelized subtasks of the UMAP algorithm. 42

4.1 Speedups of Numba compared to compiled C Code via CPython. 44
4.2 Flawed visualizations of the MNIST data set. 48

5.1 Times taken by GPUMAP subtasks on reduced GoogleNews data sets. . . 51
5.2 Comparison of visualizations by GPUMAP and UMAP on Iris and COIL-20. 53
5.3 Comparison of visualizations by GPUMAP and UMAP on MNIST, F-MNIST and GoogleNews. 54

1 Introduction

1.1 Background and Motivation

In the fields of neural networks, deep-learning and data science data with very high dimensionality can often be encountered. To recognize patterns and clusters in such data a human perceivable representation is desired. However, due to the curse of dimensionality, visualization of complex data is difficult. Two and three dimensional depictions are easiest for humans to perceive. A transformation of complex data into human understandable form is desirable.

Dimensionality reduction algorithms can be used to solve this problem. They process complex high-dimensional data into simplified versions with a lower dimensional complexity. Clearly not all information included in the data can be preserved, some will be lost along the compression. Thus, reduction will usually work best when the data itself is not as complex as the space it is embedded into. For example, a circle embedded in five dimensions cannot easily be visualized as is. A good two dimensional representation of the five dimensional data would expose the circular shape. The underlying shapes and forms that are embedded in complex spaces can be seen as manifolds on which the data points are arranged. Under the aspect of visualization quality, representing these manifolds as accurately as possible is an important trait.

Multiple dimensionality reduction algorithms with different approaches and varying qualities exist. t-SNE [1] is the current state of the art algorithm. This can for example be seen in the huge variety of existing implementations in a multitude of languages, some of which are listed on the creator van der Maaten's website[1].

t-SNE provides very good graphical results, which is the reason for its widespread adoption for visualizations in respective fields. The biggest downside is that it suffers from slow performance when compared to other dimensionality reduction algorithms, like PCA. Multiple attempts to improve this have been made. The approaches that try to parallelize t-SNE for GPUs are described in the next section. They were an inspiration for this thesis' aim to parallelize the UMAP algorithm.

Uniform Manifold Approximation and Projection, or UMAP for short, is a new dimensionality reduction algorithm, introduced in 2018 by McInnes et. al. [2]. It is building on mathematical principles of Riemannian geometry and algebraic topology. This ground-up algorithm conception allows for multiple benefits, described in Section 2.2.3. However,

[1] `https://lvdmaaten.github.io/tsne`, accessed 20.04.2019.

most prominently the paper shows that UMAP delivers graphical results as good or better than those of t-SNE, while performing faster. Since no parallelized version is available as of yet, the paper only compares sequential implementations.

This thesis' purpose is to explore the possibilities of parallelizing the UMAP algorithm. GPUs are made for massive parallel processing, allowing for a big number of calculations to be done simultaneously. And even though GPUs have various drawbacks, they usually outperform normal CPUs due to their sheer amount of processing units. Therefore a parallelized implementation of UMAP will bring the biggest performance gain, when done for GPUs.

Such an implementation is advantageous for various use cases. The enhanced performance would allow for faster visualization of complex data in general. Another motivational point is the idea of visualizing neural networks. If the performance is good enough, the implementation could be used to provide quick regular updates on their evolutions. After each interactive training phase the changes of the network's layers could be analyzed, without prolonged periods of waiting. A parallel implementation of UMAP for GPUs could make this possible.

1.2 Related Works

Multiple algorithms have already been analyzed and adapted, in order to run on GPUs. Thus, for porting UMAP an analysis of similar algorithms' ports is useful. Analyzing their approaches will provide inspiration for technologies that can be used and estimations of the speedups that can be achieved through them. t-SNE is a highly related algorithm, that also shares computational steps with UMAP. Since there are ports of t-SNE for GPUs, the following section evaluates these efforts. Listed are summarizations of two publications and explanations of their approaches. A detailed description of t-SNE is given in Section 2.2.2.

1.2.1 Linear tSNE Optimization for the Web

In [3] Pezzotti et al. develop a variation of the t-SNE algorithm that can be run in the browser. Their code[2] is based on the TensorFlow.js library[3] and available under the Apache free software license. Instead of directly porting the algorithm to run in the browser via Javascript, the publication describes how to modify t-SNE in a way that it can be run on GPUs. This way the advantage of the web's platform independence can be combined with the fast performance of GPUs.

Modern browsers are sandboxed, meaning that they prevent any executed code from directly accessing the hardware. The only way scripts can communicate with other parts

[2] `https://github.com/tensorflow/tfjs-tsne`, accessed 20.04.2019
[3] https://github.com/tensorflow/tfjs, accessed 20.04.2019

of the system is through designated APIs. WebGL [4] provides such an API, with which images can be rendered on the GPU from within the browser. These images are computed by shaders, programs that contain instructions on how to generate the graphical output. The shader code is not limited in computation complexity. Merely the input data is limited by a maximum size of textures that the GPU can hold in its main memory.

Traditionally WebGL is used for rendering. But if an algorithm can be rewritten to a form where its instructions are written as a shader and its input data can be encoded in a texture, then the algorithm can be deployed to the GPU by using WebGL. Pezzotti et al. used this to run t-SNE on GPUs from within a browser.

However, t-SNE in its normal form cannot efficiently be written as a shader, since it uses a tree data structure. Trees do not store the data they contain in a consecutive manner, but instead nested on different branches. Conditions need to be evaluated in order to know under which branch the desired data is stored. Accessing data in parallel can therefore only be done if the processing units can execute code independent of each other. GPUs are not capable of such branched executions, since all computing units need to follow the same instructions in a SIMD manner (Single Instruction, Multiple Data).

Hence Pezzotti et al. construct a variation of the t-SNE algorithm. Details on how this is achieved can be found in the publication [3]. In summarization the algorithm's modified version replaces t-SNE's tree data structure with two dimensional fields. These are represented by respective two dimensional textures and can be accessed in parallel. By sampling the textures through a shader, parallel execution of the algorithm is accomplished.

This variant of t-SNE has a time complexity of $\mathcal{O}(N)$, since the computational complexity of the sampling is linear. The linearization allows for better scaling of the algorithm, by using less main memory. But the sampling also leads to inaccuracies, since it approximates the values of the original t-SNE algorithm. Albeit the publication argues that the introduced errors are negligible, it unfortunately does not provide any results to support this. Neither execution times of the algorithm, nor graphical results are provided. Therefore a meaningful comparison with other implementations cannot be made at the time of writing.

An online example[4] using the algorithm exists. It allows to visualize parts of the MNIST data set [5]. After the initialization phase every iteration step of the t-SNE algorithm can be seen. It provides a simple user interface to configure perplexity and the amount of iterations. Figure 1.1 shows the outcome when the example is run with the default settings.

By measuring the GPU workload, an approximation on the running time was made. On a system with an Intel® Core™ i5-6200U CPU, which includes an Intel® HD Graphics 520 GPU, an execution of the algorithm took about 55 seconds. The first part, in which the nearest neighbors for each data point are computed, took approximately 20 seconds.

[4] https://storage.googleapis.com/tfjs-examples/tsne-mnist-canvas/dist/index.html, accessed 20.04.2019

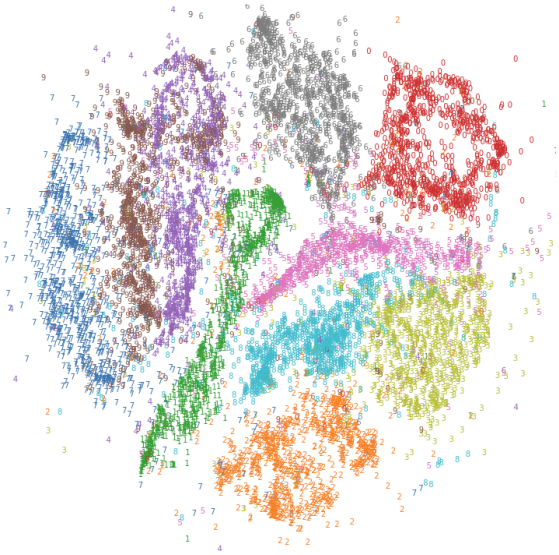


Figure 1.1: Graphical outcome of the online example of Pezzotti et al.'s t-SNE variation. The parameters used are a perplexity of 30, 800 KNN iterations and 500 t-SNE iterations. Visualized is a subset of the MNIST data set, consisting of 10.000 digits.

The GPU usage was at 100 percent throughout this part. The second part, consisting of the embedding optimization iterations, took about 35 seconds, with an average GPU usage of 50 percent.

When assuming that this performance can be linearly scaled to the complete MNIST data set, then this would outperform CPU implementations, which range between 500 and 4500 seconds according to [6]. It should be noted that the online example also provides visual feedback on the embedding for each step. These visual updates cause computational overhead through interrupts. They explain the lowered average GPU usage in the second part.

Overall this publication has good approaches. The usage of web technology makes the implementation platform-independent and convenient to use. But when compared with other technologies, such as the upcoming t-SNE-CUDA implementation, WebGL and shaders do not seem to be the best choice to maximize the computation speed.

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

As indicated by its name, t-SNE-CUDA by Chan et al. [6] uses the Nvidia CUDA technology [7] to port the t-SNE algorithm to GPUs. The code is available at the

project's Github repository[5] and is licensed under the BSD free software license. Provided results show that it outperforms all available CPU versions of t-SNE significantly. The publication claims a 50 times speedup over the fastest parallel CPU version and a 650 times speedup over a commonly used version.

The t-SNE algorithm consists of two parts. First, a K-Nearest-Neighbor (KNN) search is performed. From it a sparse representation of similarities between data points are created, based upon which data points are close to one another. In the second part a low-dimensional representation of the original data is formed. By measuring the similarities of the low-dimensional representation and comparing them to the similarities of the original input data, the low-dimensional representation is iteratively altered to represent the original data as accurately as possible.

While the original t-SNE algorithm uses a tree data structure to perform a KNN search, CUDA-t-SNE uses the index-based approximate KNN algorithm provided by the FAISS library [8]. The library's algorithm runs on the GPU and outperforms other KNN search implementations.

From the KNN search constructed data point similarities are stored in a sparse representation. Most similarities are neglected, as they have too little impact on the outcome of the algorithm. Only a certain amount of nearest-neighbors are considered for each point. In consequence the similarities can be stored in a sparse representation, resulting in a linear storage requirement. For computations on these sparse representations CUDA-t-SNE uses the cuBLAS Library [9]. cuBLAS offers parallelized versions of common BLAS (Basic Linear Algebra Subprograms) operations, such as basic matrix and vector calculations.

In the iterative second part of the algorithm, t-SNE uses the Barnes-Hut algorithm [10] to reduce the number of necessary computations between data points. In each iteration a quadtree of the current embedding is constructed. The tree allows to summarize distance calculations from one point to a group of other points, if the group is close together and has a high distance from the point. The construction of the tree is adapted from an implementation of the Barnes-Hut algorithm in CUDA [11]. Overall this part of the algorithm is analogous to the original algorithm. All operations are either on the tree or the sparse representations and are executed in parallel on the GPU.

t-SNE-CUDA can profit from building upon existing programs and libraries. Not only does this safe development time and resources, but also the existing implementations have already been tested and optimized. Thus, t-SNE-CUDA can benefit from their performance, which can be seen in the results. The speedups compared to CPU versions are high and the graphical results are indistinguishable from those of the original sequential algorithm.

[5] https://github.com/CannyLab/tsne-cuda, accessed 20.04.2019

1.2.3 Comparison of TensorFlow t-SNE and t-SNE-CUDA

The previously listed works use two very different approaches on how to use GPUs to parallelize t-SNE. They diverge on many aspects, as the rundown in the following table shows. Additionally two variants of t-SNE are added for comparison. One is part of the popular scikit-learn library [12] and is a Python implementation that only supports sequential execution. The other, Multicore t-SNE [13], is a parallel implementation, that can run on multiple CPU cores. The results listed in Table 1.1 were achieved by using up to 4 CPU cores in parallel. The performance column consists of times that the algorithm took to embed the full MNIST data set.

Table 1.1: Performance comparison of different t-SNE implementations.

Implementation	Technology	Frontend	Performance
scikit-learn t-SNE[6]	Python	Python / CLI	4556 seconds, see [6]
Multicore t-SNE	C++, Python	Python / CLI	501 seconds, see [6]
TensorFlow t-SNE	WebGL	Browser / GUI	300 seconds (estimation)
t-SNE-CUDA	CUDA	Python / CLI	7 seconds, see [6]

1.3 Aim of the Thesis

In this thesis the UMAP algorithm is to be implemented in a parallel manner, so that it can be run on GPUs. The parallel implementation will be derived from the original sequential algorithm.

Due to the limitations that come with the form of a thesis, certain constraints are imposed. While the original implementation of UMAP supports a wide range of distance metrics, the parallel implementation will exclusively focus on the Euclidean distance. UMAP builds upon multiple components that carry out subtasks of the algorithm. These subtasks will be solved by reusing existing implementations when possible. For complex components without existing GPU implementations, such as Random Projection Trees, substitute algorithms will be used. By these measures the development time is to be kept within a feasible time frame.

To implement the UMAP algorithm on GPUs the CUDA [7] technology will be used. It allows for parallel execution and memory management on GPUs. This choice is made due to CUDA's widespread usage and its good performance, as can be seen in the previous Section 1.2.2.

The developed algorithm will be evaluated. For this the overall performance will be measured, as well as the performance of the algorithm's individual steps. Paired with

[6] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html, accessed 20.04.2019

measurements of the original implementation, a comparison with the sequential version will show how much speedup is achieved by using the GPU. Analyzing the algorithm's steps independently will provide insight as to which parts of the algorithm should be considered for future optimizations.

Furthermore, a comparison with the CUDA implementation of the t-SNE algorithm will be given. It provides an estimation on the advantage of the implementation in practice.

1.3.1 Thesis Outline

The main body of this thesis is structured as follows:

- Chapter 2 describes related algorithms, for conceptual understanding of the domain. It also includes a description of the central UMAP algorithm.
- Chapter 3 covers the approach to parallelizing UMAP.
- Chapter 4 contains technical information on the implementation.
- Chapter 5 presents run time measurements of the implementation.
- Chapter 6 discusses the results and compares them with run times of other algorithms.
- Chapter 7 concludes the thesis. Suggestions on possible future works are given.

2 Preliminaries

The following sections will give an overview on clustering algorithms and dimensionality reduction algorithms. For each algorithm a description and a general implementation overview in pseudo code will be given. Furthermore, the time complexity of each algorithm is explained. Eventually, since parallelization is a central element of this thesis, each algorithm is also accompanied by a brief analysis on how it can be parallelized.

2.1 Clustering Algorithms

Clustering algorithms identify dense areas in data and group data points in such areas into clusters. They are used for numerous tasks, across multiple fields. In the context of this thesis the biggest interest is in their utilization for visual aid. While clustering algorithms do not visualize data themselves, they can be used in combination with visualization methods. Figure 2.1[1] shows an exemplary selection of visualization techniques and how they can be combined with clustering approaches.

Clustering algorithms share approaches and subroutines with dimensionality reduction algorithms. They are used in the same domain, so exploring them allows for broader insight into the field of data analysis. The following subsections covers K-Means and DBSCAN, two algorithms that are commonly used in practice. The algorithms' approaches on how to cluster points are distinct, yielding from different concepts of clusters.

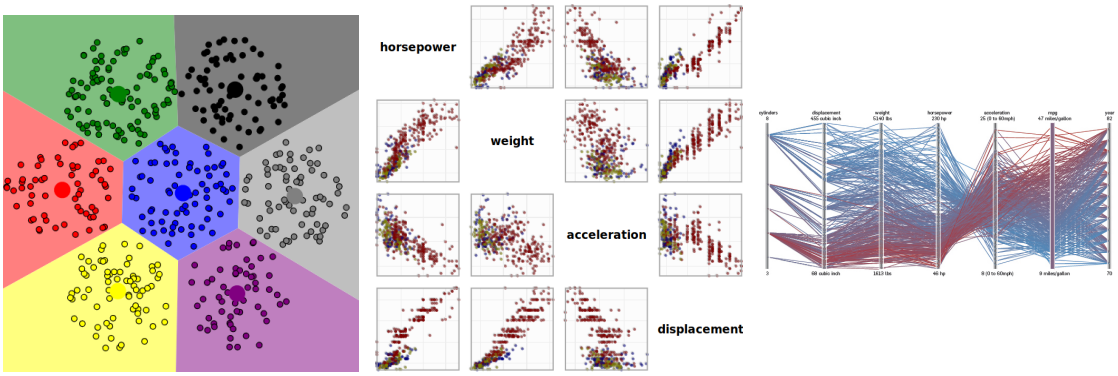
2.1.1 K-Means

K-Means, also known as Lloyd's algorithm, is a clustering algorithm publically published in 1982 [15]. It is a standard algorithm for finding clusters and is frequently used in machine learning and related fields. Common libraries used in these fields include implementations of the K-Means algorithm, i.e. tensorflow[2] and scikit-learn.[3]

[1] Figure 2.1a was created with an online tool for vizualizing how the K-Means algorithm functions (https://www.naftaliharris.com/blog/visualizing-k-means-clustering/, accessed 20.04.2019). Figure 2.1b and Figure 2.1c are taken from [14]. Their original sources are respectively https://www.naftaliharris.com/blog/visualizing-k-means-clustering/, accessed 20.04.2019 and https://homes.cs.washington.edu/~jheer//files/zoo/ex/stats/parallel.html, accessed 20.04.2019.

[2] https://www.tensorflow.org/api_docs/python/tf/contrib/factorization/KMeans, accessed 20.04.2019

[3] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html, accessed 20.04.2019



(a) Example of how clustering can support two dimensional visualization. Found clusters can be used to add color and separations between data points, resulting in clearer visual distinction of clusters.

(b) Data points in scatter plot matrices can be colored according to the cluster they belong to, giving a better intuition on how to interpret the plot.

(c) Analogous to (b) the lines of a parallel coordinates plot can be colored to ease interpretation. This improves the traceability of coordinates across the plot.

Figure 2.1: Examples of combining clustering algorithms with visualization techniques.

K-Means is based on a centroid approach to clusters. It expects a cluster to be representable through its centroid, the mean of all points belonging to the cluster. This intuitive concept of clusters defines the task of finding all clusters to finding the center of each cluster. (For a different concept of clusters see Section 2.1.2.) If the amount of clusters is known beforehand, then a converging algorithm such as K-Means can be constructed.

Starting with a randomized initialization for all K centroids $\mathbf{C}$, K-Means assigns each M-dimensional data point $\mathbf{x}_i$ of an input set $\mathbf{X}$ to the centroid μ_j closest to it. As a distance measure the Euclidean metric is used. After every point was processed, the clusters' centers are recalculated. New centroids are calculated by averaging over the positions of all data points belonging to a cluster. The previous steps are repeated until the algorithm eventually converges. Convergence has been reached when the centroids are no longer moved. The result is a desired partitioning of the data into K clusters $\mathbf{C} = f_{\texttt{K-Means}}(\mathbf{X})$.

Algorithm 1 shows a pseudo code version of K-Means. Figure 2.2 shows how the algorithm converges by repeating the steps of assigning points to the closest cluster centroid and repositioning the centroids according to the points they represent.

Algorithm 1: K-Means

Input : data set $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$,
amount of clusters K.

Output: set of clusters, defined via their centroids $\mathbf{C} = \mu_1, \ldots, \mu_K$.

```
randomly initialize the K cluster centroids C = μ_1, ..., μ_K
do
    foreach x_i ∈ X do
        find cluster μ_j with minimal distance ||x_i − μ_K||
        assign x_i to μ_j
    foreach μ_i ∈ C do
        recalculate μ_i by averaging all points belonging to it
while centroid positions changed in iteration; but no more than I times
```

The algorithm has a time complexity of $\mathcal{O}(K \cdot N \cdot M)$. This complexity stems from inside the main loop. For each of the N data points the Euclidean distance to each of the K clusters is calculated, by summing the positional values of all M dimensions. The repositioning of the K clusters has a lower complexity of $\mathcal{O}(N \cdot M)$ since each data point is only used in the calculation of its respective centroids' new mean. I, the amount of iterations which the main algorithm loop is executed, is not included in the time complexity. As stated in [16], p. 364, I can be fixed "in most cases", as the clusters found by K-Means only undergo negligible changes after a certain amount of steps. Overall, K-Means is linear in the amount of clusters, number of data points and data dimensions.

While K-Means is commonly fast due to this linear nature, it also has multiple downsides. The input of K requires a user to know how many clusters the data consists of. A common solution to this problem is to run K-Means with numerous inputs of K. Another problem is that of data being clustered in unexpected, counterintuitive ways. This problem often arises when data has clusters of varying size or density. Several variants such as K-Means++ [17] try to solve this problem by improving the initialization of the centroids. The theoretical problem that K-Means does not always find the optimal cluster assignment is still unavoidable. As shown in [18], finding clusters "in general dimension is NP-hard for $K = 2$". Thus, a linear algorithm can only ever provide an approximation.

Parallelization of K-Means can be achieved straightforward. The assignment step, where data points are assigned to clusters, can be computed independently for each data point. Similarly the repositioning of the centroids can be done in parallel, since the centroid positions are independent of one another. In between these steps synchronization is needed. Parallelized implementations using MPI and OpenMP[4] exist, as well as a CUDA version that uses data parallelism.[5]

[4] http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html, accessed 20.04.2019
[5] https://github.com/serban/kmeans, accessed 20.04.2019

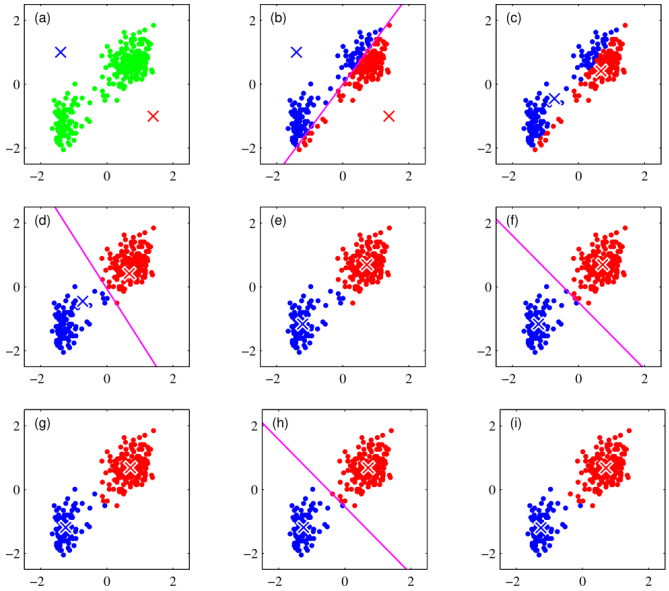


Figure 2.2: Convergence of the K-Means algorithm. (a) starts by defining two arbitrary cluster centers. The remaining steps show the alternation between assigning the points to the cluster nearest to them (indicated by a line) and recalculating the cluster centers. The figure is taken from [19], Figure 9.1.

2.1.2 DBSCAN

One of the biggest disadvantages of the K-Means algorithm is that it relies on the assumption of all clusters having one single center. This effectively prevents discovery of shapes other than spherical, especially non-convex forms.

DBSCAN, short for "Density Based Spatial Clustering of Applications with Noise" was introduced in 1996 by Ester et. al. [20]. Instead of assigning each data point to a center, the algorithm takes a different approach. It considers clusters to be dense point clouds. Figure 2.3[6] shows examples of what shapes clusters recognized by DBSCAN can have.

If a data point has more than a certain amount of points close to it, DBSCAN defines that area as dense. All points inside this dense area form a cluster. A cluster is discovered by starting from a data point $\mathbf{x}_i \in \mathbf{X}$ with at least $minPts$ other points in its ε-neighborhood. For the point and it's neighbors $\mathcal{N}(\mathbf{x}_i)$ a new cluster c_j[7] is created. All neighboring points that also fulfill the criteria of having enough points in their own neighborhood are used to expand the cluster by adding these neighbors. Through repeating this procedure, clusters of arbitrary shapes can be found, given that they are connected and fulfill the density constraint.

[6]The graphic of Figure 2.3 was taken from https://www.datanovia.com/en/lessons/dbscan-density-based-clustering-essentials/, accessed 20.04.2019

[7]DBSCAN does not define clusters by their centroids, therefore the notation is c_j, not μ_j.

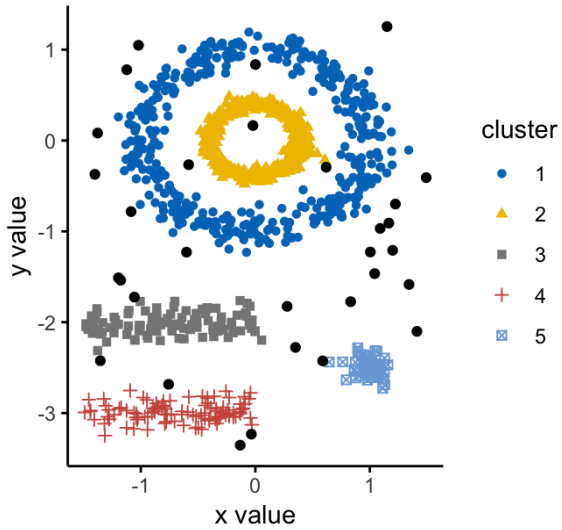


Figure 2.3: Example of non-convex clusters found by DBSCAN.

Algorithm 2 shows an outline of how the algorithm can be implemented. The result is a desired partitioning of the data into a set of clusters $\mathbf{C} = f_{\texttt{DBSCAN}}(\mathbf{X})$. The amount of found clusters is unknown beforehand. It is determined only by the input parameters and the structure of the input data.

Algorithm 2: DBSCAN

Input : data set $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$,
maximum radius for cluster connectedness ε,
minimum amount of points needed for a dense region *minPts*.

Output: a set of clusters $\mathbf{C} = c_1, c_2, \ldots$.

```
for previously unprocessed data point x_i ∈ X do
    retrieve neighbors of x_i within an ε radius N(x_i)
    if N(x_i) has more than minPts then
        create new cluster c_j consisting of x_i and N(x_i)
        initialize a new search set S = N(x_i)
        foreach s ∈ S do
            retrieve neighbors of s within an ε radius N(s)
            if N(s) has more than minPts then
                add N(s) to cluster c_j and search set S
    else
        mark x_i as noise
```

The original publication [20] claims the time complexity of DBSCAN is $\mathcal{O}(N \cdot \log(N))$, at least in practice. This can be achieved by the usage of R*-Trees for spatial queries, so

that the neighbor search can be done in $\mathcal{O}(\log(N))$ on average. Since every point is only processed once, this leads to the claimed average complexity. As the R*-Trees are the only additional data structure used, the space complexity of DBSCAN is equal to theirs, $\mathcal{O}(N)$.

As stated in [21], the worst case runtime complexity of DBSCAN still remains $\mathcal{O}(N^2)$[8]. Furthermore, it is shown that the best algorithm to solve the same problem as DBSCAN in three or higher dimensional space cannot be faster than $\Omega(N^{(4/3)})$. The publication presents an approximating variant of DBSCAN which has a guaranteed time complexity of $\mathcal{O}(N)$ and which yields results are equal to those of DBSCAN within a certain margin of error.

The way DBSCAN constructs clusters, expanding them by adding one point after another, makes it difficult to parallelize the algorithm. For a parallelization this inherently sequential data processing needs to be done in a distributed manner. Multiple parallel variants of DBSCAN exist, solving this problem in different ways. [22] gives a broad overview on previous approaches, before continuing to provide its own variant, HPDBSCAN.

With HPDBSCAN, Götz et al. propose to use a spatial data structure to partition the input data. These partitions are distributed to be searched for clusters independently, the found clusters are merged and checked for uniqueness. The results show that an implementation of this approach scales well, enabling a next to linear speedup.

In general, most parallel algorithm variants of DBSCAN scatter the input data among multiple processors, where it is searched for clusters in parallel. Afterwards found clusters are merged, depending on the way the algorithm variant is designed. Certain variants use set unions to merge found clusters, others use a master slave model, where the merging is done on only one computing unit. Overall, the sequential data access problem of the sequential version can be overcome with these strategies.

[8] This time complexity still neglects the dimensions of the data set to cluster. A factor of M can be added for $\mathcal{O}(N^2 \cdot M)$, but is, in contrary to K-Means, commonly skipped in literature.

2.2 Dimensionality Reduction Algorithms

"Visual exploration is an essential component of data analysis" as van der Maaten et. al. [23] put it. As seen in the previous section, classic visualization techniques, such as scatter plots and parallel coordinates, can be used for visual analysis of low-dimensional data. But for data with hundreds or thousands of dimensions these methods cannot effectively be applied anymore. Such high-dimensional data is common in fields as genomics, medicine and machine learning. Exploratory data analysis is nonetheless important in these fields, therefore a way to visualize the data is desirable. Here, dimensionality reduction algorithms can aid, by creating low-dimensional representations of high-dimensional data. In general dimensionality reduction algorithms can be useful for a multitude of applications, however, this thesis only takes interest in using them as means of visualization.

In the following sections the commonly used algorithms PCA and t-SNE will be covered. Furthermore the central algorithm of the thesis, UMAP, will be introduced. PCA is a linear transformation, that preserves the global structure of its input data, whereas t-SNE and UMAP are non-linear dimensionality reduction methods, that preserve the local structure of the input data. Especially UMAP tries to preserve both, global and local structure alike. To highlight the difference between the graphical results of the algorithms, a comparison of them is given as a conclusion to the chapter.

2.2.1 PCA

One of the most commonly used dimensionality reduction algorithms is Principal Components Analysis (PCA). Its origins are "difficult to trace" according to [24]. They lie within the beginning of the 20th century, but throughout history, PCA has been discovered multiple times, which is an indicator of how fundamental the algorithm is. It is a versatile algorithm with multiple applications, that is still frequently referenced and used in current research publications. A common application in the field of data analysis is that of visualizing data.

PCA is a statistical method to reduce data dimensionality. It creates a linear transformation of the data utilizing the data's variance. In simple terms, PCA finds a set of vectors that describe the dimensions in which the data is the most spread out. These vectors are the principal components of the data.

If the input data is interpreted as a matrix $\mathbf{X} = (\mathbf{x}_{ij})$, then the principal components of $\mathbf{X}$ are equivalent to its eigenvectors. They are independent of each other, as all eigenvectors are orthogonal to one another. The eigenvalues λ_i that correspond to the eigenvectors $\overline{\mathbf{e}}_i$, describe how high the data's variance along the respective eigenvector is.

Sorting the eigenvectors according to their eigenvalues will result in a sorted list $\mathcal{E} = (\overline{\mathbf{e}}_1, \lambda_1), \ldots, (\overline{\mathbf{e}}_N, \lambda_N)$, in which the principal components that contribute the most to the

variance of $\mathbf{X}$ are listed first. Respectively the later entries in the list contribute the least variance and therefore also hold the least information on the nature of the data in $\mathbf{X}$.

Following the procedure, the dimensionality of $\mathbf{X}$ can be lowered by keeping only the first $\widetilde{M}$ principal components. This transformation looses as little variance of the data as possible and thus preserves as much of the data's information. While the data can also lay on a manifold embedded in a high-dimensional space, PCA is a linear dimensionality reduction algorithm and merely has insight on the global structure of the data.

PCA starts by centering the input data according to the mean value in each direction. The M-dimensional input data $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$ is interpreted as an $N \times M$ matrix. Each of the N data points can hereby be thought of as as an observation of M variables. A mean vector $\bar{\mathbf{m}}$ is calculated by normalizing the sum of all input data points $\bar{\mathbf{m}} = \frac{1}{N} \sum_{\mathbf{x}_i \in \mathbf{X}} \mathbf{x}_i$. The centered version of $\widetilde{\mathbf{X}} = \tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_N$ is then obtained by setting $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \bar{m}$.

In the next step the centered matrix is used to calculate the covariance matrix $\mathbf{C} = \frac{1}{N-1}\widetilde{\mathbf{X}}^T\widetilde{\mathbf{X}}$. Through diagonalizing $\mathbf{C}$, the eigenvectors $\mathbf{e}_i$ and eigenvalues λ_i are retrieved. Sorting them by decreasing eigenvalues yields $\mathcal{E} = (\mathbf{e}_1, \lambda_1), \ldots, (\mathbf{e}_N, \lambda_N)$. The first $\widetilde{M}$ eigenvectors are the sought principal components.

From the selected eigenvectors a $M \times \widetilde{M}$ matrix $\mathbf{W}$ can be constructed by using the eigenvectors as matrix columns. Finally, the matrix $\mathbf{W}$ can be used to create a low-dimensional representation $\mathbf{Y}$ for the input data $\mathbf{X}$ with $\mathbf{Y} = \mathbf{W} \cdot \mathbf{X}$.

Algorithm 3: Principal Components Analysis.

Input : M-dimensional data set $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$,
number of dimensions to reduce to $\widetilde{M}$.
Output: $\widetilde{M}$-dimensional representation $\mathbf{Y}$ of $\mathbf{X}$.

1 mean of $\mathbf{X}$, $\bar{m} \leftarrow \frac{1}{N} \sum_{\mathbf{x}_i \in \mathbf{X}} \mathbf{x}_i$
2 calculate centered matrix $\widetilde{\mathbf{X}} \leftarrow (\mathbf{x}_i - \bar{m})$.
3 covariance matrix $\mathbf{C} = \frac{1}{N-1}\widetilde{\mathbf{X}}^T \times \widetilde{\mathbf{X}}$
4 diagonalize $\mathbf{C}$ to compute the eigenvectors and eigenvalues
$\mathcal{E} = (\mathbf{e}_1, \lambda_1), \ldots, (\mathbf{e}_N, \lambda_N)$
5 sort $\mathcal{E}$ by decreasing eigenvalue
6 create $N \times \widetilde{M}$-dimensional matrix $\mathbf{W}$ using the first $\widetilde{M}$ eigenvectors of $\mathcal{E}$
7 low-dimensional representation $\mathbf{Y} = \mathbf{W} \cdot \mathbf{X}$

Calculating the covariance matrix has a time complexity of $\mathcal{O}(M^2 \cdot N)$ for a naive implementation of matrix multiplications. Additionally the diagonalization of the covariance matrix has a time complexity of $\mathcal{O}(M^3)$. Thus, this PCA variant has an overall time complexity of $\mathcal{O}(M^2 \cdot N + M^3)$.

Faster PCA variants exist. Instead of calculating the covariance matrix, these variants use either other methods to calculate the eigenvectors and -values, such as Singular Value

Decomposition (SVD), or calculate one principal component after another in an iterative process.

Parallelizing PCA can be done without any changes to the algorithm. All performed operations can be composed of BLAS (Basic Linear Algebra Subprograms) operations. They are therefore already available in a multitude of parallel libraries, such as OpenBLAS[9] or cuBLAS [9]. Andercut et al. used the cuBLAS library to implement a GPU version of a PCA variant in [25] and received a 12 times speedup over their parallel CPU implementation.

2.2.2 t-SNE

t-SNE is the state of the art dimensionality reduction algorithm for visualizing high-dimensional data. It was introduced by Van der Maaten et al. in 2008 [1]. The name is derived from Stochastic Neighbor Embedding (SNE) [26], of which t-SNE is a variation. The t stems from its usage of the Student's t-distribution, one of several modifications introduced by t-SNE.

The publication shows that t-SNE's graphical results surpass those of other related algorithms. Figure 2.4 provides an example for this, a comparison between visualizations on the basis of the COIL-20 data set [27]. The COIL-20 data set is composed of picture series from 20 different objects, each photographed from 72 equidistant angles by a camera orbiting the object. As can be seen, t-SNE captures the circular nature of the data well. It correctly represents the data of several objects as closed loops, while the algorithms used for comparison can at best display the data in a curved form.

In 2014 a refined version, Barnes-Hut t-SNE, was published by van der Maaten [23]. It reduces the runtime of the algorithm significantly, while providing the same quality of graphical output. Since there are no real differences for users of the algorithm, the Barnes-Hut version is commonly referred to by t-SNE. To keep its terminology aligned, this thesis does the same. The following algorithm description also focuses on the refined version, while still describing the differences between the two.

The basic concept of t-SNE is to find a low-dimensional representation of the input data, that preserves the structure of the input data as well as possible. Thus, a way to measure the quality of such a low-dimensional representation is needed. For this t-SNE uses the Kullback-Leibler divergence $f_{KL}()$. From both, the original data $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$ and the low-dimensional representation $\mathbf{Y} = y_1, \ldots, y_N$, probability distributions are determined. These probability distributions describe the affinities between all points in the respective data. $P = (p_{ij})$ contains all affinities of points $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$, $Q = (q_{ij})$ all of $y_i, y_j \in \mathbf{Y}$.

P and Q are calculated based on a distance function $dist()$, to capture the layout of the data. By default t-SNE uses the Euclidean metric, but any other distance function can

[9] http://www.openblas.net, accessed 20.04.2019

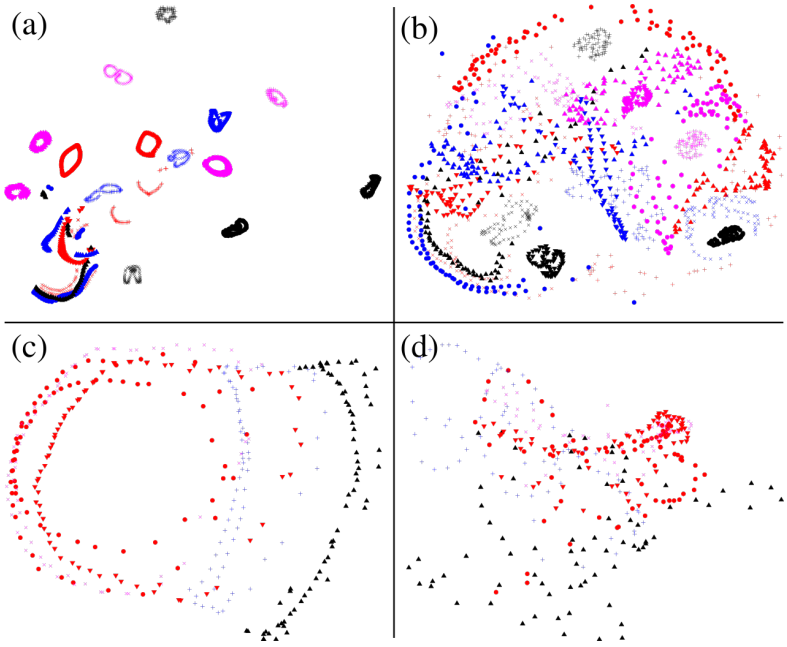


Figure 2.4: Visualizations of the COIL-20 data set [27] by a) t-SNE, b) Sammon mapping, c) Isomap and d) LLE. The graphic is taken from [1], Figure 5.

be used too. Data points that have a short distance from one another are assigned a high affinity, while those further apart from each other receive a low affinity.

In the same manner that every point $\mathbf{x}_i \in \mathbf{X}$ has a low-dimensional representation $y_i \in \mathbf{Y}$, every p_{ij} responds to a q_{ij}. Comparing p_{ij} with q_{ij} shows how well the affinity between $\mathbf{x}_i$ and $\mathbf{x}_j$ is preserved by their low-dimensional counterparts y_i and y_j. This comparison is done with the Kullback-Leibler divergence. Using it on all pairs of $p_{ij} \in P$ and $q_{ij} \in Q$, the quality of the low-dimensional representation is assessed. As a consequence a cost function $C = f_{KL}(P||Q) = \sum_i \sum_j p_{ij} \log(\frac{p_{ij}}{q_{ij}})$ can be formulated. The lower the Kullback-Leibler divergence between the probability distribution, the better the found representation $\mathbf{Y}$.

t-SNE minimizes C by using the gradients $\frac{\delta \mathcal{C}}{\delta y_i}$. Each gradient describes how C changes in relation to the respective y_i. Using a learning rate parameter η and a time-dependent momentum parameter $\alpha(t)$, t-SNE improves $\mathbf{Y}$ with a gradient descent procedure, according to the formula $\mathbf{Y}^{(t)} = \mathbf{Y}^{(t-1)} + \eta \frac{\delta \mathcal{C}}{\delta \mathbf{Y}} + \alpha(t)(\mathbf{Y}^{(t-1)} - \mathbf{Y}^{(t-2)})$.

This fundamental approach of t-SNE is the same for both, the original and the Barnes-Hut variant. The two big improvements made by the Barnes-Hut variant are a) neglect of infinitesimal probabilities when calculating the probability distributions p_{ij} and b) summarizing similar probability distributions when calculating q_{ij}, to speed up the gradient calculation. Both are achieved by using tree data structures.

Through the usage of a vantage-point tree, as introduced in [28], a K-Nearest-Neighbors (KNN) search is performed. For each point $\mathbf{x}_i$ only the K found nearest-neighbors $\mathcal{N}_K(\mathbf{x}_i)$ are used to calculate the probability distributions. All other probability distributions with $\mathbf{x}_j \notin \mathcal{N}_K(\mathbf{x}_i)$ are ignored by defining $p_{ij} = 0$. These nulled affinities can be omitted when storing P by using a sparse data structure, reducing the otherwise needed $\mathcal{O}(N^2)$ space complexity to a linear $\mathcal{O}(N)$.

The K parameter of the KNN search is provided by using a multiple of the user-defined perplexity input parameter *perp*. "The perplexity can be interpreted as a smooth measure of the effective number of neighbors.", as is stated in [1]. So the choice of *perp* should be made based on the amount of neighbors that a data point is expected to have on average, "typical values are between 5 and 50". The perplexity parameter was already present in the original t-SNE variant, despite the lack of such a KNN search. Instead it was and still is used to normalize the probability distributions of P. To simplify the algorithm description, further details on the parameter are omitted here, they can be found in the publication [1].

The low-dimensional representation $\mathbf{Y}$ is initialized randomly. Through the gradient descent procedure described above, $\mathbf{Y}$ is shaped to better represent $\mathbf{X}$. A majority of the probability distributions q_{ij}, upon which this procedure builds, are small, resulting from many points having a big distance from one another. Individually these q_{ij} contribute very little to the gradient $\frac{\delta \mathcal{C}}{\delta \mathbf{Y}}$. It is desirable not to calculate all of them, in order to accelerate the algorithm. However, while they are negligible on their own, the amount of such q_{ij} is too high to discount all.

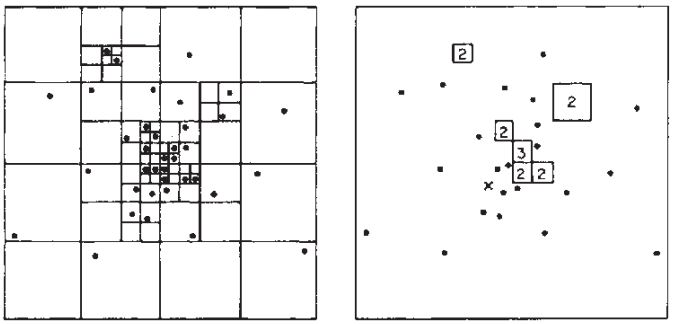


Figure 2.5: The quadtree structure shown on the left side is used by the Barnes-Hut algorithm to group data points as shown on the right side. Distances from the starting point, the x mark, can be calculated faster this way. The graphic is taken from [10], Figure 1.

Therefore, t-SNE utilizes the Barnes-Hut algorithm [10]. It creates a spatial tree, e.g. a quad tree for two dimensional space, and uses it to summarize distances between data points that are far apart. If a point has approximately the same distance to a group of other points, this group will be treated as a single point, using the average of the group's positions to calculate the distance. This average is pre-calculated for each space partition in the tree and does not cause overhead per query. Figure 2.5 shows an example of how data points are grouped with the tree structure. Calculating the distances between all points can hereby be sped up, since fewer calculations are needed.

Algorithm 4: (Barnes-Hut) t-Distributed Stochastic Neighbor Embedding

Input : M-dimensional data set $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$, number of dimensions to reduce to $\widetilde{M}$, perplexity *perp*, number of iterations T, learning rate η, momentum $\alpha(t)$.

Output: $\widetilde{M}$-dimensional representation $\mathbf{Y}$ of $\mathbf{X}$.

1 **foreach** $\mathbf{x}_i \in \mathbf{X}$ **do**
2 using vantage-point trees, find KNN $\mathcal{N}_K(\mathbf{x}_i)$ with $K = \lfloor 3\ perp \rfloor$
3 compute symmetric pairwise affinities p_{ij} for all $\mathbf{x}_j \in \mathcal{N}_K(\mathbf{x}_i)$
4 sample initial low-dimensional representation $\mathbf{Y}^{(0)} = \mathbf{y}_1, \ldots, \mathbf{y}_n$ from $\mathcal{N}(0, 10^{-4}I)$
5 **for** $t = 1$ *to* T **do**
6 calculate low-dimensional affinities q_{ij} of $\mathbf{Y}^{(t)}$ using the Barnes-Hut algorithm
7 compute gradient $\frac{\delta \mathcal{C}}{\delta \mathbf{Y}}$ with these q_{ij}
8 $\mathbf{Y}^{(t)} \leftarrow \mathbf{Y}^{(t-1)} + \eta \frac{\delta \mathcal{C}}{\delta \mathbf{Y}} + \alpha(t)(\mathbf{Y}^{(t-1)} - \mathbf{Y}^{(t-2)})$

Both tree data structures, the vantage-point tree and the quadtree used by the Barnes-Hut algorithm, and the sparse representation, to store the p_{ij} in, require $\mathcal{O}(N)$ storage. This is therefore also the overall storage requirement of Barnes-Hut t-SNE. This is an improvement over the original t-SNE algorithm, which had a space complexity of $\mathcal{O}(N^2)$, required to store all probability distributions p_{ij}.

The time complexity of Barnes-Hut similarly improved from the original $\mathcal{O}(N^2)$ to $\mathcal{O}(N \cdot \log(N))$. All operations on trees are either $\mathcal{O}(1)$ or $\mathcal{O}(\log(N))$ and thus result in a combined complexity of $\mathcal{O}(N \cdot \log(N))$ when repeated for all of the N input data points. Other factors, such as the dimensionality M and the number of iterations T, can be considered part of the input size N or a constant factor respectively, and are not considered in the runtime complexity of t-SNE.

Parallelizations of t-SNE are commonly in use. Besides the GPU parallelizations covered in the related works Section 1.2, there are also parallel CPU implementations. The best performing one is Multicore t-SNE [13], an adoption of the source code released alongside the original Barnes-Hut t-SNE publication.[10] The repository states, that the biggest speedup can be made through the parallelization of the KNN search. In general parallelizing the used algorithms, KNN search and Barnes-Hut, is a key aspect.

[10] https://github.com/lvdmaaten/bhtsne, accessed 20.04.2019

2.2.3 UMAP

Uniform Manifold Approximation and Projection, or UMAP for short, was introduced in 2018 by McInnes et al. [2]. The algorithm was inspired by t-SNE, as stated in [29] (minute 16:35). The intention was to create a similar dimensionality reduction algorithm that is based on a mathematical foundation. As UMAP was introduced as late as 2018, no secondary literature is available at the time of writing. The primary resources on UMAP, besides the publication itself, are the source code repository[11] and a talk of McInnes on the matter [29].

Most prominently the publication shows that UMAP delivers graphical results as good or better than those of t-SNE, while performing faster. But it also has further improvements over t-SNE:

- UMAP is "viable as a general purpose dimension reduction technique"[2], so it can be used for more than just visualization. t-SNE performs badly when reducing to more than two or three dimensions, due to the spatial data structures it uses.
- UMAP allows for the usage of user-defined metrics with which the input data is interpreted.
- UMAP can add additional data to a previously created low-dimensional representation.

The detailed math and derivation of the algorithm will not be discussed here, since it is of no interest to this thesis' goals. Instead an analysis of the performed steps and the necessary computations will be done. The primary focus is on explaining how the mathematical concepts are implemented, so that the analysis of their parallelization in the following chapter is comprehensible.

Conceptually UMAP takes a similar approach to dimensionality reduction as t-SNE. It first creates an abstract model of the given data $\mathbf{X}$. Then it initializes an $\widetilde{M}$-dimensional representation $\mathbf{Y}$. Now $\mathbf{Y}$ is iteratively shaped according to the abstract model of $\mathbf{X}$. Eventually $\mathbf{Y}$ will be a good low-dimensional representation of $\mathbf{X}$, that not only preserves the local structure, but also separates unlike data points, creating clusters thereby.

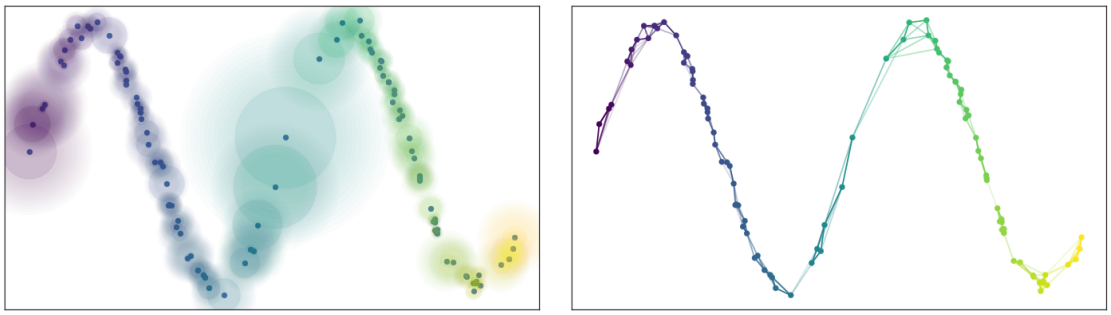
Rather then using a probability distribution like t-SNE, UMAP uses a weighted graph as its abstract model. The motivation as well as the justification for this stems from the mathematical background of UMAP. Summarized, the publication shows that the structure of the input data can be captured by a simplicial complex, if the data's manifold is approximated via a local connectivity assumption. In other words, when creating a cover of the data via nearest-neighbor relations, the data's topology is represented independent of the N-dimensional space in which it is embedded.

[11] https://github.com/lmcinnes/umap, accessed 20.04.2019

The simplicial complex can hereby be thought of as a graph. It actually is a set of connected simplices, but only 0-simplices and 1-simplices are considered. Since a 0-simplex is simply a representation of a data point and a 1-simplex is a connection between two 0-simplices, they are equivalent to nodes and edges of a graph. For easier comprehensibility the upcoming algorithm description will therefore simply refer to the simplicial complex as a graph.

UMAP starts by performing a KNN search on the input data $\mathbf{X}$. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is created, each data point $\mathbf{x}_i \in \mathbf{X}$ is represented by a node $v_i \in \mathcal{V}$. For each of the K found nearest-neighbors an edge e between the respective v_i and v_j is inserted in $\mathcal{E}$. e is assigned a weight $w_{i,j} = \overline{dist}(\mathbf{x}_i, \mathbf{x}_j)$, which can be considered a measure as to how important the edge is.

The function $\overline{dist}$ hereby normalizes the distance between two points $\mathbf{x}_i$ and $\mathbf{x}_j$ according to the distance of $\mathbf{x}_i$ to its closest neighbor. It is locally normalized in a way that the sum $\sum_{e=(v_i,)\in\mathcal{E}} weight(e)$ over all edge weights is the same for each node. This assures that the contribution of each node to the structure of the graph is the same, a constraint that is necessary for the mathematical background to hold. Figure 2.6 gives an intuition on the involved distance functions and why they need to be normalized individually per node. Figure 2.7 shows how such a locally normalized distance function can help overcome the curse of dimensionality.

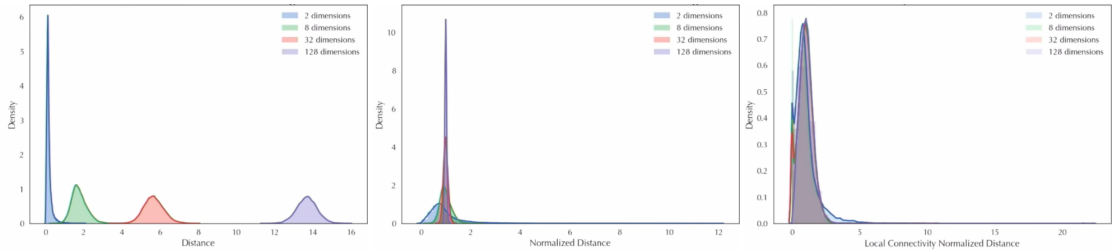


(a) Visualization of the distances created by UMAP through a KNN search. The distance to the first node of the KNN is shown as the radius of a plain circle. The distances to the remaining KNN are of descending importance, visualized by a blur surrounding the nodes. Graphic taken from [29] (minute 11:11).

(b) The graph created from the locally normalized distances. The line thickness visualizes the weight of edges. Due to the normalization areas with few data points, such as the center in this example, are still well captured and strongly connected. Graphic taken from [29] (minute 16:35).

Figure 2.6: Exemplary graph created by UMAP for data lying on a sine wave manifold.

$\mathcal{G}$ is now used to initialize the $\widetilde{M}$-dimensional representation $\mathbf{Y}$. It is created by calculating the eigenvectors of the adjacency matrix for $\mathcal{G}$. The $\widetilde{M}$ eigenvectors with the highest eigenvalues form $\mathbf{Y}$.



(a) Distances to 20 nearest-neighbors of data randomly sampled from a normal distribution. Graphic taken from [29] (minute 11:30).

(b) The same distances, normalized. All high-dimensional data points have appropriately the same distance from one another. This is also known as the curse of dimensionality. Graphic taken from [29] (minute 11:50).

(c) The same distances, normalized by local connectivity as in Figure 2.6a. The result is a normal distance distribution, independent of the number of dimensions. Graphic taken from [29] (minute 12:10).

Figure 2.7: Overcoming the curse of dimensionality with locally normalized distances.

With this initial low-dimensional representation $\mathbf{Y}$ in place, UMAP proceeds to improve $\mathbf{Y}$. Using a stochastic gradient descent (SGD) technique the points $\mathbf{y}_i \in \mathbf{Y}$ are repositioned over the course of T iterations. The function to minimize is $\Phi(\mathbf{y}_i, \mathbf{y}_j) = (1 + a(\|\mathbf{y}_i - \mathbf{y}_j\|_2^2)^b)^{-1}$. a and b are hereby precomputed constants, chosen to define a differentiable function. The gradient along Φ is used to update $\mathbf{y}_i$. For each edge $(v_i, v_j) \in \mathcal{E}$ an update is performed, so that $\mathbf{y}_i \leftarrow \mathbf{y}_i + \alpha \cdot \nabla(log(\Phi))(\mathbf{y}_i, \mathbf{y}_j)$ where α is a scaling factor that decreases proportional to the amount of passed iterations.

Since a SGD and not a normal gradient descent procedure is used, such updates are not performed for all edges in each iteration. The edges are sampled by only considering them for an update with a certain likelihood. The weight of the edge determines the probability with which it is sampled. In the case that it is sampled an additional negative sampling step is also performed. For that, a set of randomly selected points Neg is used to additionally reposition $\mathbf{y}_i \leftarrow \mathbf{y}_i + \alpha \cdot \nabla(1 - log(\Phi))(\mathbf{y}_i, \mathbf{y}_k)$ where $\mathbf{y}_k \in Neg$.

While the regular SGD step is meant to correctly position $\mathbf{y}_i$ and $\mathbf{y}_j$ in relation to each other, the negative sampling step increases the distance between unrelated data points. Thus, the regular step draws related points closer together, which forms clusters in the low-dimensional representation, whereas the negative sampling step pushes points further away from each other, creating gaps in between these clusters. This spreading out of the data is one of the reasons for UMAP's improved graphical results when used for visualization, since it preserves the global structure of the original data.

Algorithm 5: Uniform Manifold Approximation and Projection

Input : M-dimensional data set $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$, number of dimensions to reduce to $\widetilde{M}$, number of neighbors to consider K, number of iterations T.

Output: $\widetilde{M}$-dimensional representation $\mathbf{Y}$ of $\mathbf{X}$.

```
Initialize graph G = (V, E) with V = v_1, ..., v_N and E = ∅
foreach x_i ∈ X do
    find KNN N_K(x_i)
    foreach x_j ∈ N_K(x_i) do
        w_{i,j} ← dist(x_i, x_j), where dist is the locally normalized distance
        E ← E ∪ e = (v_i, v_j) with weight(e) = w_{i,j}
initialize Y = y_1, ..., y_N with M̃ eigenvectors from the adjacency matrix of G
for n = 1 to T do
    foreach e = (i, j) ∈ E do
        if weight(e) > r, where r ∈ (0, 1) is a random value then
            use gradient-descent to reposition y_i with respect to
              a) y_j, the representation of e's other node, and
              b) a set of randomly selected points for negative sampling
```

The used KNN search algorithm, NN-Descent, is an approximate KNN algorithm with a stated "empirical complexity of $\mathcal{O}((N \cdot M)^{1.14})$" [30]. The stochastic gradient descent with the negative sampling has a time complexity of $\mathcal{O}(K \cdot N)$". Since $K \ll N$ the total time complexity is bound by the KNN search. Analogous to the time complexity of t-SNE, the amount of iterations T is considered a constant. The space complexity of UMAP can be dominated by either the graph structure or the low-dimensional representation, depending on whether $\widetilde{M}$ or K is bigger. It therefore is $\mathcal{O}((K + \widetilde{M}) \cdot N)$.

An analysis of how UMAP can be parallelized is given in Chapter 3.

2.2.4 Visual Results

To compare the visualization results of the aforementioned dimensionality reduction algorithms, Figure 2.8 shows visualizations for several data sets. The MNIST data set [5], is commonly used for assessing the visualization quality of dimensionality reduction algorithms. By reference to it, it can be seen that the quality of PCA's visualization is inferior to those of t-SNE and UMAP. The points in the graphical output of PCA are merely spread out, while t-SNE and UMAP both show clearly recognizable clusters. The colors representing the labels of the data set show that the clusters are correctly formed and consist of similar data. The differences between t-SNE and UMAP are not quite as grave, but nonetheless very noticeable. The clusters formed by t-SNE are closer together with more noise in between and not as dense as those created by UMAP. Additionally UMAP produces no split up clusters as in the lower part of t-SNE's visualization.

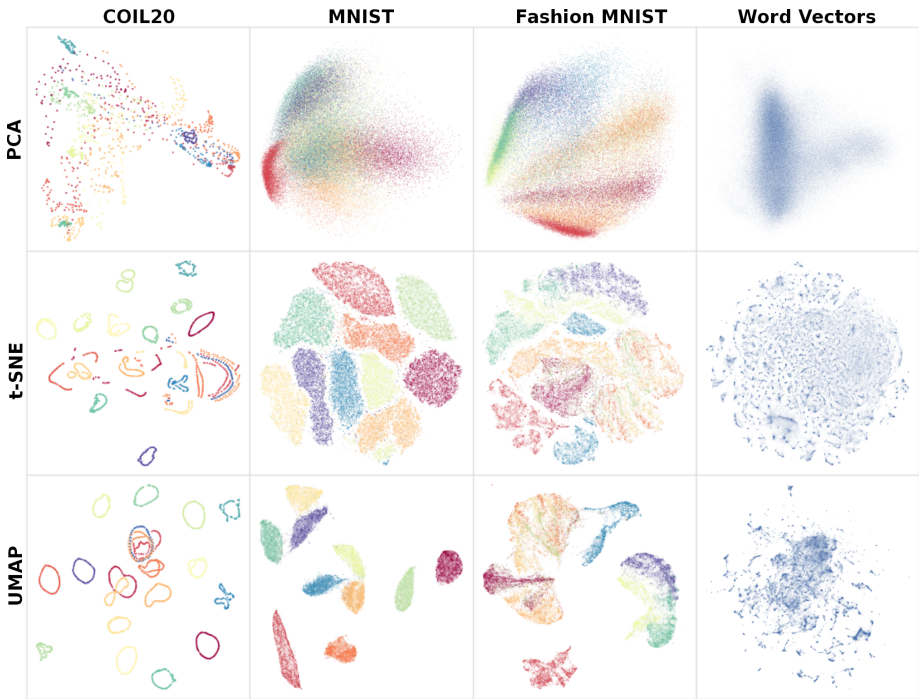


Figure 2.8: Visualizations of various data sets by PCA, t-SNE and UMAP. The figure is modified from the UMAP publication [2], Figure 2.

2.3 Algorithm Complexity Overview

The algorithms of this chapter all have varying complexities for both required time to execute the algorithm, as well as space required by it. To recapitulate, Table 2.1 gives a condensed overview of these complexities. As before, N denotes the amount of data points in the input data and M denotes their dimensionality. Additionally, for dimensionality reduction algorithms $\widetilde{M}$ denotes the amount of dimensions to reduce to.

Table 2.1: Comparison of Time and Space Complexities for presented Algorithms.

Algorithm	Time Complexity	Space Complexity	Notes
K-Means	$\mathcal{O}(K \cdot N \cdot M)$	$\mathcal{O}(N \cdot M)$	K number of clusters to find
DBSCAN	$\mathcal{O}(N^2 \cdot M)$	$\mathcal{O}(N)$	
PCA	$\mathcal{O}(M^2 \cdot N + M^3)$	$\mathcal{O}(N \cdot M)$	
t-SNE	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	M contained in N
Barnes-Hut t-SNE	$\mathcal{O}(N \cdot log(N))$	$\mathcal{O}(N)$	M contained in N
UMAP	$\mathcal{O}((N \cdot M)^{1.14})$	$\mathcal{O}((K + \widetilde{M}) \cdot N)$	K for the KNN algorithm

3 Methods

This chapter is comprised of three parts. In the first, UMAP is profiled in order to determine which parts of the algorithm should be parallelized. In the second, approaches to parallelizing the found algorithm steps are given. The last section concludes the chapter with a mapping of the parallelized algorithm steps onto the pseudo code of UMAP.

3.1 Profiling UMAP

This section first identifies UMAP's subtasks and profiles what impact on the running time they have. Thereafter, the influence of the data set size and dimensionality on these times is explored. Finally, the expected variance of measured times is estimated by repeated executions.

3.1.1 Identifying Subtasks of UMAP

To efficiently parallelize an algorithm, it is important to know how well the individual steps of the algorithm perform. Parallelizing a part that hardly contributes to the algorithm's running time, will not result in a noticeable speedup. By profiling the algorithm, the parts that take longest to execute can be found. A parallelization of these parts will have the biggest impact on the performance of the algorithm.

UMAP is a general dimensionality reduction algorithm and is thus applicable to a multitude of data sets. Throughout the following profiling a wide selection of data sets is used, in order to capture how the performance varies for different inputs. The data sets are: the Iris flower data set [31], the Pen Digits data set [32], the COIL-20 [27] and COIL-100 [33] data sets, the Labeled Faces in the Wild (LFW) data set [34], the MNIST data set [5], the Fashion-MNIST (F-MNIST) data set [35], the CIFAR-10 data set [36] and two subsets of, aswell as the full GoogleNews word vectors data set [37]. For each data set the number of samples N and dimensionality M is given in the profiling table. The majority of these data sets are also used to measure algorithm performances in the original UMAP publication [2]. Since this thesis focuses on using UMAP for visualization, all data sets are reduced to two dimensions. All UMAP input parameters are set to the implementation's default values.

Before processing, all data sets were made continuous in the main memory, meaning that they were placed consecutively in memory without any fragmentation. Initial test profilings did not assure the data to be continuous, which resulted in implausibly high execution times. In these test runs a visualization of the MNIST data was performed 5 times slower. A processing of the CIFAR-10 data set even took 25 times as long as the continuous version.

Profiling is performed on a system with two 2.3 GHz Intel® Xeon® Gold 5118 CPUs (24 cores) and 256 GB of main memory. UMAP in version 0.3.8 is used along with Numba version 0.43.1. The operating system is Ubuntu 16.04 with kernel version 4.4.0 and Python 3.7.3.

The standard Python profiler[1] is used for accurate timing. It measures the time spent on each function of the program. This enables the identification of program parts that take long to execute and therefore are good candidates for parallelization.

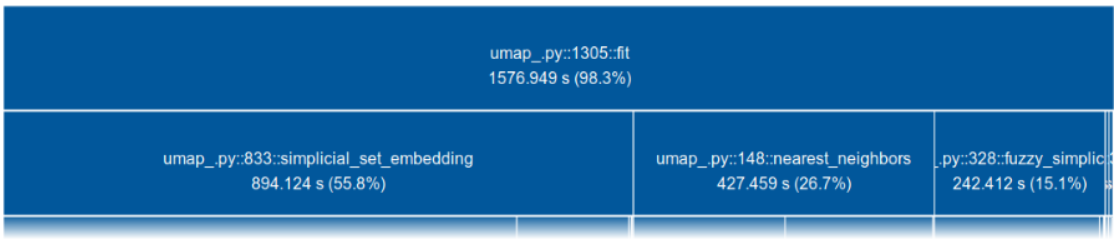


Figure 3.1: Profiling of UMAP on a 500.000 points subset of GoogleNews.

Figure 3.1 shows a visualization of profiling data generated by processing a 500.000 points subset of the GoogleNews data set. The graphic is taken from the interactive output that was generated by the `tuna` Python package[2] for the profiling data. As shown, the total runtime is dominated by few functions. The percentages in the graphic do not sum up to 100%, because they include loading times of the data set and libraries. Nonetheless, more than half of the time is spent in the `simplicial_set_embedding` subtask. It performs the initialization and optimization of the low-dimensional representation. Other significant contributions are made by the `nearest_neighbors` subtask, which performs a KNN search, and the `fuzzy_simplicial_set` subtask, which normalizes distances between data points found by the KNN search (as illustrated in Figure 2.6).

UMAP uses Numba [38], a Just-In-Time (JIT) compiler. It compiles source code "on the fly" during execution of the program. Consequently the time taken for these compilations also accounts for parts of the run time. This time is not prominently present in Figure 3.1, since it is comparatively low and compiling is done at several occasions during the execution.

[1] https://docs.python.org/3/library/profile.html, accessed 25.04.2019
[2] https://pypi.org/project/tuna/, accessed 25.04.2019

In Table 3.1, the times measured in the UMAP publication (Publ.) are compared to the times of the profiling (Total). The latter is further broken down into time spent on each of the dominating subtasks, `simplicial_set_embedding` (Embed), `nearest_neighbors` (KNN) and `fuzzy_simplicial_set` (Fuzzy), along with accumulated time spent on compiling (JIT).

Table 3.1: Profiling of UMAP on various data sets.

Data set	N	M	Publ.	Total	KNN	Fuzzy	Embed	JIT
Iris	150	4		9.26	0.01	6.31	2.58	8.76
COIL-20	1440	16384	12	13.10	0.09	6.77	5.64	8.48
Digits	1797	64	9	14.49	0.18	7.19	6.87	8.74
LFW	13233	2914		47.84	15.98	12.16	18.64	11.64
COIL-100	7200	49152	85	90.86	55.59	9.66	21.61	11.89
MNIST	70 K	784	87	162	34.24	39.10	86.00	11.70
F-MNIST	70 K	784	65	164	35.44	38.85	87.58	11.57
CIFAR	60 K	3072		200	79.84	34.34	82.30	11.29
GoogleNews	200 K	300	361	585	142	110	326	11.96
GoogleNews	500 K	300		1577	427	242	894	11.45
GoogleNews	3 M	300		14226	6277	1614	6249	12.04

The table shows differences between the times of the publication and the profiling. This is mostly due to the difference in hardware being used. Most parts of UMAP are sequential, therefore the times of the publication, that were measured on an ordinary consumer CPU, outperform the server hardware used for the profiling, since the publication used a CPU with a higher clock rate. It can be noticed that the difference is smaller for data sets with high dimensionality. This could be caused by recent changes to the implementation or those data sets requiring more computations that are parallelized for UMAP.

For the full GoogleNews data set, the KNN search and the embedding subtask take equally long, but for most smaller data sets the embedding subtasks dominates. An exception to this is the COIL-100 data set, here the KNN subtask requires more than twice as long as the embedding. It seems that the KNN subtask is strongly influenced by the dimensionality of the data. The differences in JIT times are due to an optimization of the implementation, which uses a non-compiled KNN algorithm for small data sets.

3.1.2 Influence of Data Set Size and Dimensionality

To estimate the influence of the data set's number of samples and dimensions on the running time, a separate profiling is performed. UMAP is hereby applied to reduced versions of the CIFAR data set. A reduction is made in equidistant steps for both, the data set size and dimensionality. The size is reduced in steps of 10.000 data points, the

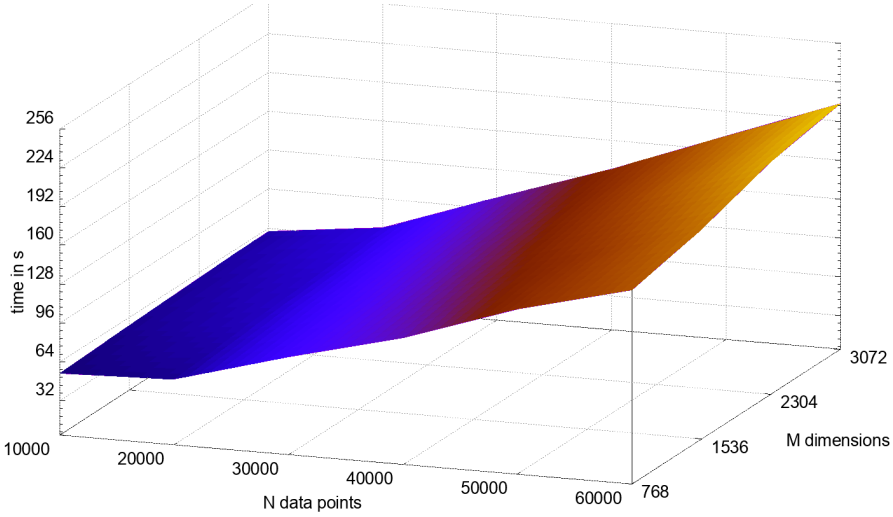


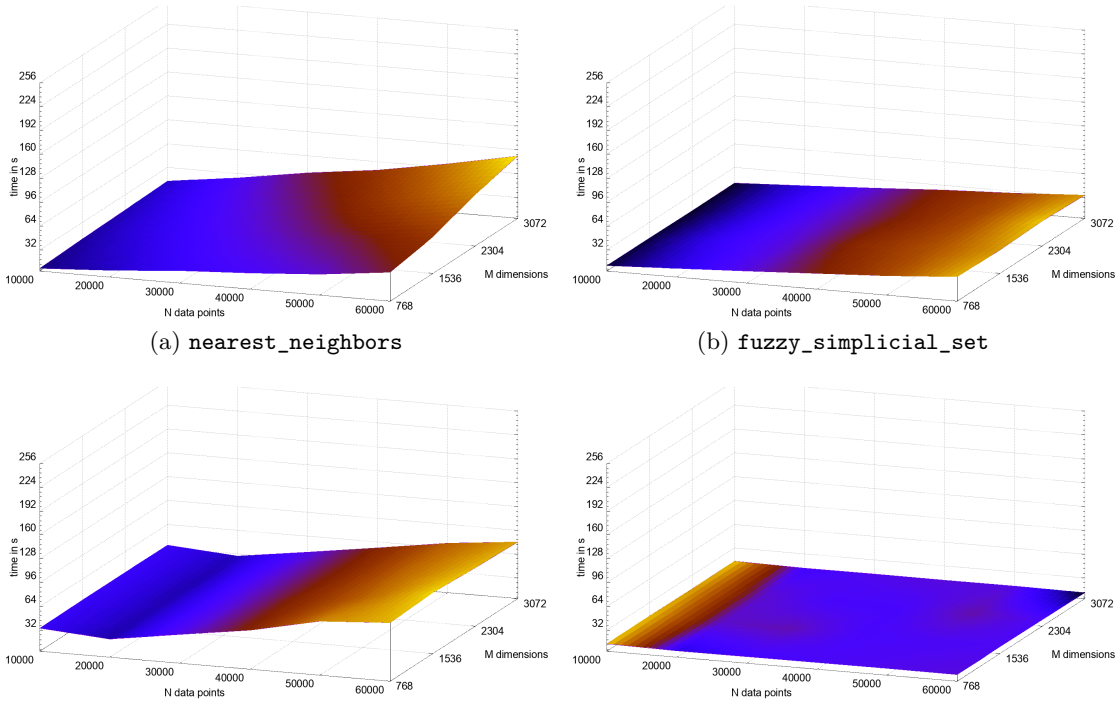
Figure 3.2: Profiling of UMAP on varying sizes and dimensions of CIFAR.

dimensionality is reduced in steps of 25% of the original number of dimensions. This is done by dropping rows and columns of the data set respectively. CIFAR was chosen, because it is big enough to allow for the creation of meaningful subsets through reduction, while still being small enough to be processed repetitively in a feasible time frame.

As the visualized profiling results of Figure 3.2 show, the total running time of UMAP is stronger influenced by the amount of samples, than by the dimensionality. Processing only half of all data points reduces the running time more, than processing the same amount of data points with only a fourth of the dimensions. Figure 3.3 shows how this is reflected in the individual subtasks of UMAP. It shows the contribution of each subtask that resulted in the total running time of Figure 3.2.

Only the `nearest_neighbors` subtask is affected by the dimensionality of the data. The `simplicial_set_embedding` subtask solely computes on the low-dimensional embedding and thus is independent of the dimensionality. It shows linear growth with the data set size. The `fuzzy_simplicial_set` subtask has the same properties, albeit growing slower. It normalizes all distances between nearest-neighbors, which are stored in a sparse matrix. Thus it is not affected by the dimensionality either. JIT compiling always takes approximately the same amount of time and is not affected by the input data.

A comparison of these time measurements is given in Figure 3.4. Here, solely the data set size has been altered, while the original dimensionality was kept. The drop in time of the `simplicial_set_embedding` subtask is caused by UMAP performing a lower amount of embedding optimization iterations for data sets bigger than 100.000 data points.



(a) `nearest_neighbors`

(b) `fuzzy_simplicial_set`

(c) `simplicial_set_embedding`

(d) JIT compiling

Figure 3.3: Time spent by individual subtasks processing reduced CIFAR data sets.

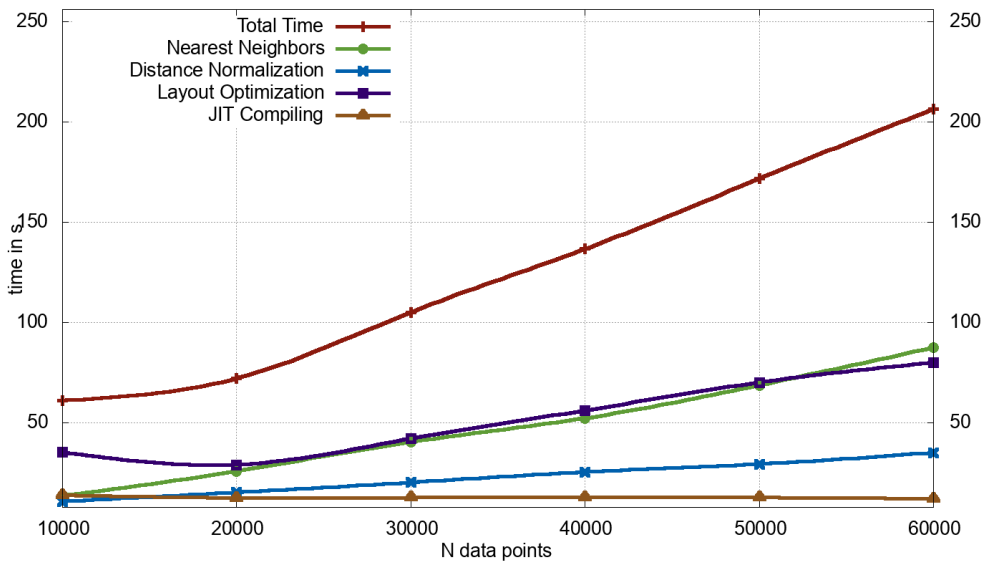


Figure 3.4: Comparison of UMAP subtasks on reduced CIFAR data sets.

3.1.3 Variation of Execution Time

To analyze how deterministic the UMAP algorithm behaves, a final profiling is done. Here, UMAP was performed on subsets of the CIFAR data set a total of 10 times. From the measured execution times the standard deviation is calculated. Figure 3.5 displays the average execution time with the standard deviation displayed as a span on each measured data point. The variation seems to grow linearly with the total execution time. This suggests that the runtime of UMAP is rather deterministic and only varies within a certain range of expected execution time.

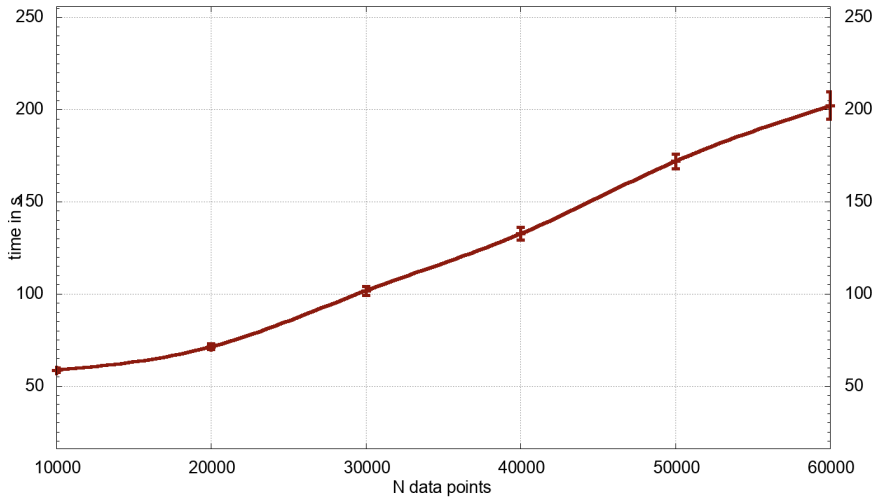


Figure 3.5: Standard deviation of UMAP execution times on reduced CIFAR data sets.

3.2 Parallelizing Subtasks of UMAP

From the results of the profiling it is clear which algorithm steps take the longest. Ordered by position in the algorithm these subtasks are: performance of a KNN search, normalization of distance between data points and initialization and optimization of a low-dimensional representation. In the following each step is analyzed to determine how it can be parallelized for GPUs.

3.2.1 Parallelization of KNN Search

Finding nearest-neighbors is a common problem and encountered in a wide range of fields, e.g. Machine Learning and Data Analysis. As a consequence, multiple implementations already exist. Comparing the performance of these implementations based on publications alone is difficult. Provided times depend on the systems used for measurement.

In [39] a framework for benchmarking approximate KNN algorithms is introduced. It compares a selection of KNN implementations in a standardized and reproducible manner. The implementations are required to provide a Python interface, in order to be compared by the framework. This allows them to be used as a drop-in replacement for the current `nearest_neighbors` subtask of UMAP.

Unfortunately all but one of the compared implementations of [39] only support execution on CPUs. Since the goal of this thesis is to parallelize UMAP on GPUs, only the GPU algorithm is to be considered further. The implementation in question is the FAISS library [8], which is also used by t-SNE-CUDA (see Section 1.2.2). In the benchmarking framework only the CPU version of FAISS is compared, which commonly performs worse than the remaining algorithms.

However, the GPU version of FAISS is shown to perform well in t-SNE-CUDA [6]. The use case of KNN search in t-SNE is similar to that in UMAP, which is querying a fixed amount of nearest-neighbors for each data point. Thus, the performance of utilizing FAISS for UMAP can be expected to be of similar quality. Conclusively FAISS will be used for a GPU implementation of the `nearest_neighbors` subtask.

3.2.2 Parallelization of Distance Normalization

Normalizing the distances between data points by their local connectivity is a task unique to UMAP and not provided by any library. Therefore a new implementation for GPUs is needed. Figure 3.6 shows the time taken by the `fuzzy_simplicial_set` method, when UMAP is processing a big subset of the GoogleNews data set. It can be seen that most of the time, besides JIT compiling, is spent in a call to the `smooth_knn_dist` method.

Parallelizing `smooth_knn_dist` can be done in a straightforward manner, since the method is "embarrassingly parallel": it processes every data point independently. This is advantageous for creation of a parallel version, as no extra caution is required to avoid data conflicts.

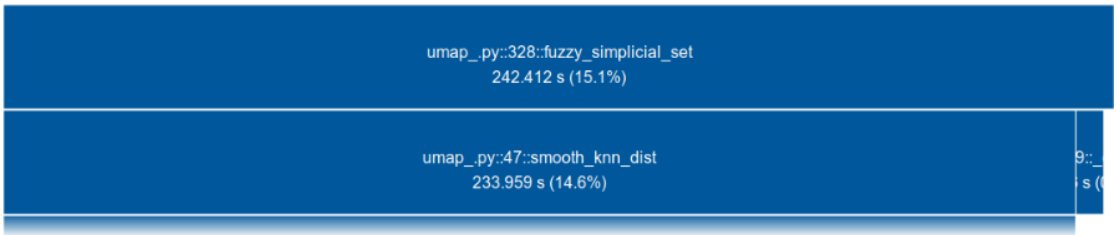


Figure 3.6: Time spent by `fuzzy_simplicial_set` when processing 500.000 points of the GoogleNews data set.

3.2.3 Parallelization of Embedding Initialization

Figure 3.7 shows that most time spent by `simplicial_set_embedding` is spent on a call to the `optimize_layout` subtask. The remaining time is mostly taken up by a call to the `spectral_layout` subtask. As can be seen in Figure 3.8, the time spent on the latter increases disproportionately with increasing data set size.

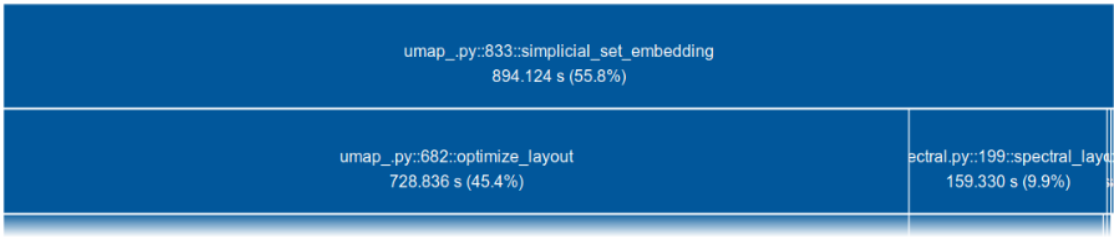


Figure 3.7: Time spent by `simplicial_set_embedding` when processing a 500.000 points subset of GoogleNews.

`spectral_layout` initializes the low-dimensional representation, by calculating eigenvectors and -values of the sparse adjacency matrix induced by the nearest neighbors. Each KNN relation is represented as an edge in a graph. With additional processing of neighborhood relations, the matrix, albeit being represented in a sparse form, grows. Processing of the 500.000 point subset of the GoogleNews data set, with a neighborhood value of $K = 15$ for the KNN search, results in a graph that consists of 13.8 million edges. Calculating all eigenvalues and eigenvectors for this matrix consequently takes a long time.

The spectral initialization allows "faster convergence and greater stability" ([2], p. 14), but the embedding also "can be initialized randomly", as the publication states. Parallelizing the `spectral_layout` method is difficult, since it requires a multitude of numerical operations on sparse matrices and handling of edge cases, such as disconnected graphs. Considering this, a parallelization of the method is avoided. Instead, spectral initialization is only used for small data sets. For data sets bigger than a certain threshold a random initialization is performed. Considering Figure 3.8, a limit of 100.000 data points is chosen.

3.2.4 Parallelization of Embedding Optimization

The remaining time of `simplicial_set_embedding` is mostly accounted for by the call to `optimize_layout`. It is the essential part of the UMAP algorithm, that creates the low-dimensional representation. Given a pre-initialized embedding, generated with or without `spectral_layout`, a Stochastic Gradient Descent (SGD) procedure is performed. Thereby, the given embedding is reshaped to better represent the original input data, by repositioning its points. The points are repositioned according to the original data's topology, that is represented in the form of a graph.

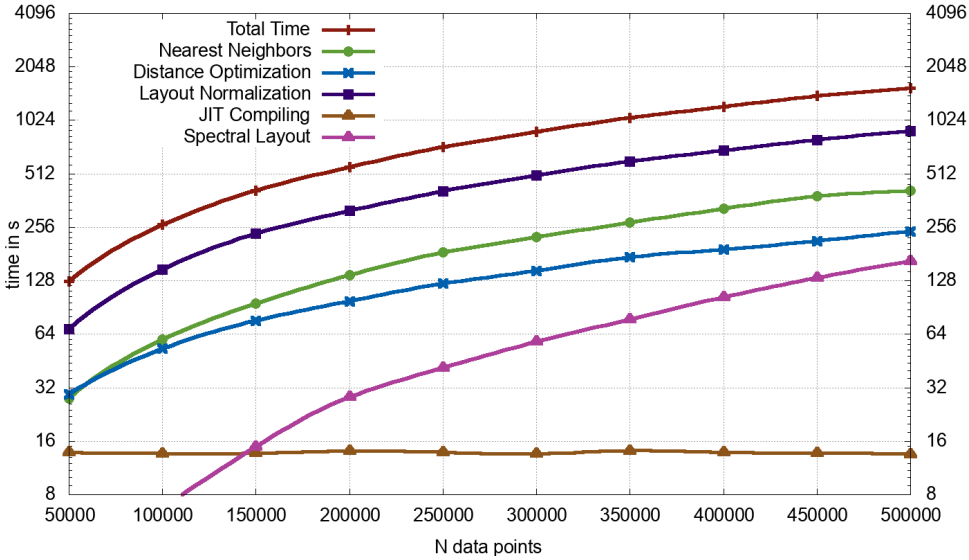


Figure 3.8: Comparison of UMAP subtasks on reduced GoogleNews data sets. The Y-axis is scaled logarithmical, so all function plots are distinguishable.

The following steps are performed for each iteration of the SGD: Each edge of the graph is iterated over and sampled with a chance based on the edge's weight. The two nodes of each edge are represented by two embedding points. If the edge is sampled in an iteration, then the distance between these two points is calculated. Depending on whether the distance reflects the weight of the edge, the embedding points are moved closer together or further apart.

Additionally negative sampling is performed for each edge with the same likeliness. For simpler implementability and better performance, this always coincides with the times that the edge is sampled. The edge's primary node is used. It is determined by the order in which the edge nodes are stored. Negative sampling randomly selects embedding points and calculates distances from them to the embedding point associated with the primary node. This primary node is then repositioned according to the calculated distances, in most cases pushing it further away from the randomly selected embedding points, as it is unlikely to be neighboring to the other points.

For the implementation of this SGD procedure conditional branching is required in multiple places, e.g. upon deciding whether or not to sample an edge. Branching however negatively impacts the performance of GPUs, since GPUs operate according to the Single Instruction Multiple Data (SIMD) model. All threads perform the same sequence of commands in parallel, while processing different data points. If the commands executed contain branching, the threads are no longer guaranteed to execute the same commands. This requires the GPU to execute every set of branched off threads separately, as only one command can be run by all threads at a time. In the worst case this can lead to sequential execution.

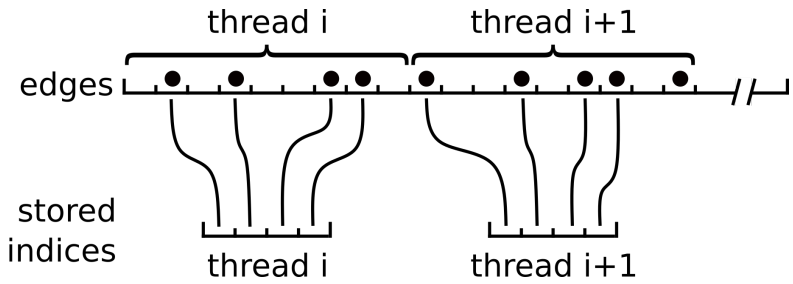


Figure 3.9: Shown is a conceptual depiction of the edge pre-filtering procedure. The edges on which SGD is to be performed are divided into blocks, each assigned to a different thread. All threads process the assigned edges in parallel. Every thread has a limited local cache. Edges that are chosen to be sampled, marked in the graphic with black dots, are stored by index in the cache. These indices are then iterated over during regular SGD, again by all threads in parallel. Since the local cache is of limited size, the procedure is done iteratively until all edges are checked for potential sampling.

One solution to reduce branching execution for SGD is, to prepare the edges which to sample in a certain iteration by pre-filtering before each operation. If every thread knows on which edges to perform SGD, then the branching introduced by the probability with which to sample an edge, can be overcome. Figure 3.9 explains this concept with a graphical depiction.

Further possibilities to improve the performance of `optimize_layout` are preventing data write conflicts by assigning nodes to threads, and improving cache performance by using shared random values per block of threads. Both concepts are more implementation focused and therefore presented in Chapter 4.2.5.

3.3 Coverage of Subtasks

Figure 3.10 shows how the identified subtasks correlate to the algorithm steps. As can be seen, the complete algorithm is covered, thus parallelizing these subtasks will lead to a fully parallelized algorithm.

Algorithm 5: Uniform Manifold Approximation and Projection

`nearest_neighbors`

1 Initialize graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = v_1, \ldots, v_N$ and $\mathcal{E} = \emptyset$
2 **foreach** $\mathbf{x}_i \in \mathbf{X}$ **do**
3 find KNN $\mathcal{N}_K(\mathbf{x}_i)$

`fuzzy_simplicial_set`

4 **foreach** $\mathbf{x}_j \in \mathcal{N}_K(\mathbf{x}_i)$ **do**
5 $w_{i,j} \leftarrow \overline{dist}(\mathbf{x}_i, \mathbf{x}_j)$, where $\overline{dist}$ is the locally normalized distance
6 $\mathcal{E} \leftarrow \mathcal{E} \cup e = (v_i, v_j)$ with $weight(e) = w_{i,j}$

`simplicial_set_embedding`

7 initialize $\mathbf{Y} = \mathbf{y}_1, \ldots, \mathbf{y}_N$ with $\widetilde{M}$ eigenvectors from the adjacency matrix of $\mathcal{G}$
8 **for** $n = 1$ *to* T **do**
9 **foreach** $e = (i, j) \in E$ **do**
10 **if** $weight(e) > r$, *where* $r \in (0, 1)$ *is a random value* **then**
11 use gradient-descent to reposition $\mathbf{y}_i$ with respect to
a) $\mathbf{y}_j$, the representation of e's other node, and
b) a set of randomly selected points for negative sampling

Figure 3.10: Parallelized subtasks of the UMAP algorithm.

4 Implementation

This chapter documents the implementation of GPU parallel UMAP (GPUMAP). Section 4.1 covers the technologies used and section 4.2 provides implementation details for the parallelized algorithm steps.

4.1 Used Technologies

To implement UMAP on GPUs, a library to communicate with the GPU is needed. As already argued in Section 1.3, the choice for this thesis is to use the CUDA library. It provides good performance, a stable API and can be widely used since the supported Nvidia® GPUs are commonly available.

To avoid an implementation from scratch, the existing UMAP Python implementation is used as groundwork. Building upon it, all subtasks that according to the analysis of Chapter 3 benefit from parallelization, will be extended to support parallel execution on GPUs. The CPU parts can still remain available and be used as a fallback if no GPU is available.

As mentioned in the previous chapter, UMAP uses Numba [38] to create compiled code during execution. This allows for performant execution of computation-heavy code, without the need to rewrite it in compiled languages such as C. The table in Figure 4.1 shows exemplary performance speedups when compared to standard Python code. The JIT compilation of Numba is not limited to CPUs, it also supports "compiling a restricted subset of Python code into CUDA kernels."[1] While not the complete API of CUDA is provided, all features required for this thesis are supported. Thus, the methods described in the following sections are implemented in Python, which is JIT compiled by Numba and executed on GPUs.

Another library used, is the `cupy` library [40]. It provides functionality to move arrays from and to the GPU memory, as well as common operations that can be performed on arrays on the GPU, such as sorting and mean value calculation. `cupy` can in many places be used as a drop-in replacement for the common `numpy`[2] library. Not only does this avoid copying from and to the GPU device, but also the operations are performed with GPU-acceleration.

[1] from http://numba.pydata.org/numba-doc/latest/cuda/overview.html, accessed 28.04.2019
[2] https://docs.scipy.org/doc/numpy/, accessed 20.05.2019

Matrix Size	Numba	C
64 x 64	463x	453x
128 x 128	454x	407x
256 x 256	280x	263x
512 x 512	276x	268x

Figure 4.1: Speedups over normal Python interpreter for naive matrix multiplication of Numba JIT and compiled C Code via CPython. Taken from [38], Table 1.

4.2 Implementation Details

Each proceeding section covers details on the implementations of one subtask for the GPU. To be easier to follow, they are sorted in the order that they occur in the algorithm. However, the first section describes what GPUMAP does if no GPU is available.

4.2.1 Fallback Routines

To be able to execute on the GPU, the GPU parallel UMAP (GPUMAP) algorithm needs to assure that a supported GPU is available. Numba provides an API to discover availability with `cuda.is_available()`. It can be queried during execution, which allows to dynamically execute either a GPU or CPU implementation. The following subtasks all use this method to allow running the CPU version as a fallback to the default GPU implementation.

For example the `nearest_neighbors` subtask of the upcoming section was modified to either call the new GPU implementation `nearest_neighbors_gpu` if a GPU is available, or the existing CPU implementation of UMAP, which was renamed to `nearest_neighbors_cpu`.

4.2.2 Subtask `nearest_neighbors`

The first step of the UMAP algorithm is a KNN search. As the FAISS library is used to handle this subtask, the new implementation is mostly a wrapper function for communication with the library. It first creates an index for the input data, on which a KNN search is to be performed, and then tasks FAISS to find the desired amount of nearest-neighbors for each data point using that index.

The chosen KNN search to be performed by FAISS is a brute-force method, since all other attempted variants performed worse. This also holds for an implementation analogous to that of t-SNE-CUDA [6]. As the non-brute-force algorithm relies on sampling, it could theoretically be made faster by using fewer samples, but this resulted in too low quality of the found nearest-neighbors, to be used further on in the algorithm.

FAISS offers a dedicated method for a brute-force KNN search, but it performed just as fast, while running out of memory sooner than the chosen version. By default FAISS reserves 18% of the GPU memory for KNN search queries. Since GPUMAP needs to retrieve all neighbors for all nodes, the whole query set will not fit into the 18% reserved memory, if the data set is bigger than the reserved memory. Therefore querying of too big data sets is done in five iterations, so that no additional memory needs to be allocated, hereby saving time.

Since FAISS only supports the use of an Euclidean metric, a check is performed that it also is the user-defined metric. Otherwise the CPU version is used. Further, FAISS does not support searching for more than 1024 nearest-neighbors, which therefore also causes a fallback to the CPU version.

4.2.3 Subtask `fuzzy_simplicial_set`

Most time spent by the `fuzzy_simplicial_set` subtask is used on a call to `smooth_knn_dist`, which iterates over the distances calculated by the previous algorithm subtask, `nearest_neighbors`. It "smooths" the distances between nearest-neighbors, by normalizing each set of edges related to one node. The normalization is based on the node's local connectivity and done independently for each set of distances, making the method "embarrassingly parallel". As a result, no memory access restrictions need to be managed.

For parallelization each thread is assigned a block of consecutive sets of distances. Each set contains the distances between one node and its KNN. By iterating over these sets, two steps are performed. First a distance with which to normalize is determined. That distance is then used in a binary search to normalize all values.

The original CPU code of the `smooth_knn_dist` contained a comment, that the code is "very inefficient"[3]. Indeed the code does contain several redundant recalculations of mean values and passes each distances set multiple times. The GPU code fixes this by calculating the mean over all values once, and calculating all other required values during a single pass of each distance set.

Further parallelization of `fuzzy_simplicial_set` was done for the `compute_membership_strengths` subtask, which is equally embarrassingly parallel and straightforward to implement. Additionally some array manipulations were performed on the GPU by using `cupy`.

[3] https://github.com/lmcinnes/umap/blob/a858c6322a3e682d8daf9c17e13ac023f3e18cfa/umap/umap_.py#L96, accessed 28.04.2019

4.2.4 Subtask `simplicial_set_embedding`

The most time of the `simplicial_set_embedding` subtask is spent on a call to the `optimize_layout` subtask. Due to its complexity it is split out into its own section. This section covers the remaining parts.

`simplicial_set_embedding` receives the normalized KNN distances from the `fuzzy_simplicial_set` subtask. It removes duplicates and values that are too small from them. This is implemented by using the `cupy` library.

Afterwards it initializes the low-dimensional embedding. For inputs with less than 100.000 data points the CPU implementation of `spectral_layout` is used. For bigger inputs a random initialization is done in parallel on the GPU.

One last step is to calculate the probabilities with which to sample in the upcoming `optimize_layout` subtask. This is handled by the `make_epochs_per_sample` subtask, which is GPU-parallelized by using cupy. Then the `optimize_layout` subtask is called. All data is already allocated on the GPU, so that no additional GPU-CPU data transfers are needed.

4.2.5 Subtask `optimize_layout`

`optimize_layout` is the implementation of the Stochastic Gradient Descent procedure. It consists of several nested loops, in which positive sampling and negative sampling are done. The most outer loop iterates over the number of epochs of the SGD. Parallelizing this loop is not possible, as each epoch builds on the processed embedding data of the previous one.

The loop nested one layer deeper iterates over all edges. It can be used for parallelization, albeit not as directly as e.g. `smooth_knn_dist`, since each edge links two separate data points. When an edge is sampled both of these points are repositioned in SGD. Since the graph is connected, each node is part of more than one edge and parsing both nodes of the edges therefore causes data access conflicts.

A solution to avoid these conflicts is to duplicate all edges, assign the associated nodes to the duplicated edge in reversed order, and then sort all edges by their first node. As it happens to be, this is already the case due to the way the KNN search builds the graph. If each edge is only sampled for its first node, and one thread is responsible for all edges where this node is the first node, then the changes to this node can be written without any data write conflicts.

When processing each edge split into its two nodes, another problem occurs. Since the nodes of the edge are processed independently, they can also read varying values due to desynchronization. While one node of an edge could be processed at the beginning of an epoch, the other could be processed at the end. The nodes will be repositioned independently of each other, which also means that the weight of the edge connecting the

nodes, could be badly represented after the epoch ends. This is caused by a less obvious read-write conflict that arises due to the data being processed in-memory. Figure 4.2b shows an example for an embedding produced with this error. Memory caches amplify the problem by providing values that are outdated, but not invalidated.

The problem can be resolved by using two embedding data arrays. All read operations are performed on one embedding, while the changes are written to the other. When all threads are done processing the edges assigned to them, the data sets are swapped and the procedure is repeated.

While this approach may work in theory, a practical problem is the mapping of nodes to threads. If the assigning is done statically, then the number of edges that one thread is in charge of, can vary widely. Balancing the computation load equally on all threads requires a sophisticated algorithm. Since a parallel algorithm would need to consider a multitude of variables, this should be done in a sequential algorithm, which would add extra time required for copying from and to the GPU.

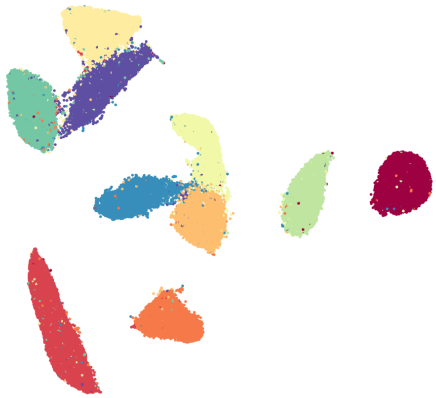
The way the GPUMAP implementation resolves this problem, is by not assigning nodes to threads. Instead the algorithm as described is performed, but the edges are statically assigned to each thread. Since only the first and the last node that a thread is responsible for, could potentially be processed by another thread, these cases are treated separately. To not overwrite potential changes by the other thread that could write to this node, each write access is done by averaging the previous and the new value. This way the loss of information is reduced and write conflicts can be mitigated.

With the core implementation in place, further optimizations can be tried out.

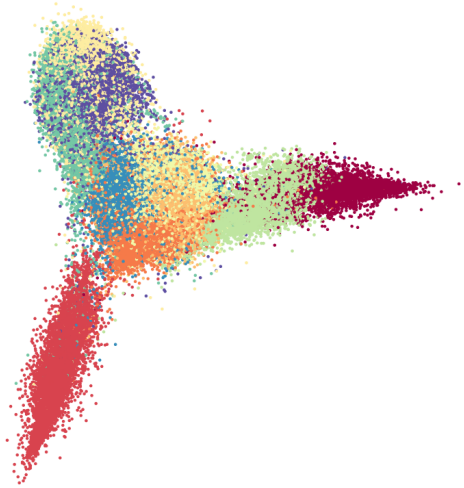
A successful optimization of the algorithm is the use of global variables. These are seen as constants by the Numba JIT compiler, allowing for constant folding and loop unrolling. A specialized implementation originally fine-tuned for the embedding into two dimensions was thus made redundant, as it now performed slower than the optimized generic version.

A less successful attempt to improve performance was by optimizing memory access. Since accessing any data in memory usually causes the hardware to load not just the desired data values, but the whole memory bank in which it resides, random accesses to memory are very inefficient and lead to many cache misses. The random accesses caused by negative sampling are thus very inefficient, as only a small part of the cached memory is actually used. If a group of threads with a shared cache were to use successive nodes for negative sampling, the caching efficiency and therefore memory efficiency could increase. The downside to this approach however, is the loss of entropy. As the result of the implementation in Figure 4.2d shows, the visualization quality deteriorates significantly, rendering this optimization impractical.

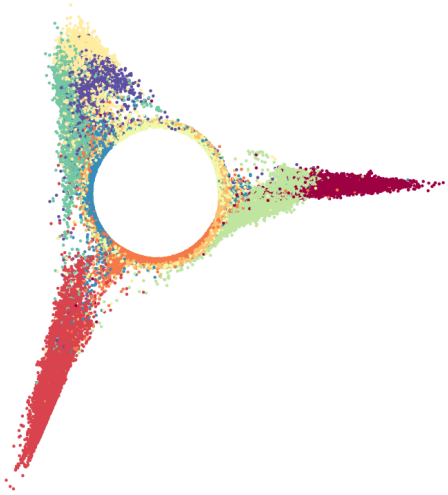
Another successful optimization was already covered in Section 3.2.3. By using a pre-filtering method, branching of the GPU threads can be avoided. During the implementation it brought a big performance gain, halving the previous time spent on



(a) Expected visualization of MNIST, given for comparison.



(b) Caused by data inconsistency when processing nodes of edges separately.



(c) Caused by thread desynchronization.

(d) Caused by using collective random values for blocks of threads.

Figure 4.2: Flawed visualizations of the MNIST data set.

`optimize_layout`. However, avoidance of branching is not just important for performance. The way the random number selection for negative sampling was originally done, caused the parallel version to look like shown in Figure 4.2c.

As expected, a random number is created. It is used to randomly access a data point of the embedding. To assure that no node of the current edge is selected, a simple `if` construct is used, that branches off and continues with the remaining loop iterations of the negative sampling. The hole that can be observed circles the zero point, so the reason could potentially be that all points were negatively sampled with zero and thus were pushed away from it.

$$k = \texttt{rand()}\%(N-1) + i + 1 \tag{4.1}$$

Replacing the branching with a modified version of the random number generation based on Equation 4.1, avoids this strange side effect. The formula can be used to receive a random point k out of N points, while guaranteeing that the value i will never be chosen for k. The formula can be nested to avoid multiple points. This way both nodes of an edge can be avoided during negative sampling.

5 Results

In this chapter the new implementation, GPUMAP, is profiled and compared to the CPU implementation and t-SNE CUDA. Section 5.1 contains performance measurements and comparisons, Section 5.2 compares the quality of GPUMAP's visualizations.

5.1 Performance

To profile GPUMAP, a similar analysis to that in Chapter 3 is performed. The hardware and software are the same, but computation power is enhanced by the usage of an Nvidia® TESLA V100™ GPU with 32 GB of Memory.

The times given in Table 5.1 show a breakdown of the total time into the times spent on the methods that were parallelized: `nearest_neighbors` (KNN), `fuzzy_simplicial_set` (Fuzzy) and `simplicial_set_embedding` (Embed). Additionally the accumulated compiling time (JIT) is listed. Further the total time (Total) of GPUMAP is given together with the speedup achieved over regular UMAP (Speedup). A visualization of how GPUMAP performs on rising numbers of data points is shown in Figure 5.1.

Table 5.1: Profiling of GPUMAP on various data sets.

Data set	N	M	Speedup	Total	KNN	Fuzzy	Embed	JIT
Iris	150	4	1.6	5.60	0.01	1.51	3.46	4.10
COIL-20	1440	16384	2.6	6.24	0.14	1.72	3.77	4.19
Digits	1797	64	2.5	5.85	0.24	1.61	3.56	4.05
LFW	13233	2914	8.1	5.91	1.24	0.86	3.13	4.08
COIL-100	7200	49152	8.7	10.40	2.57	0.78	5.62	4.09
MNIST	70 K	784	17.0	9.52	1.93	1.00	5.88	4.15
F-MNIST	70 K	784	17.2	9.73	1.90	1.07	5.98	4.24
CIFAR	60 K	3072	18.9	10.59	3.10	1.01	5.38	4.22
GoogleNews	200 K	300	61.4	9.52	3.38	1.54	3.99	4.21
GoogleNews	500 K	300	59.4	26.57	15.76	3.17	6.95	4.12
GoogleNews	3 M	300	25.7	554	502	18.32	33.16	4.20

For big data sets the total time is solely dominated by the KNN search. Compared to UMAP, the embedding optimization and fuzzy distance normalization contribute a lower

percentage of the overall running time. For data sets bigger than 100 K data points the embedding is initialized randomly, which is the reason for the drop of execution time in the Embed column. The time spent on JIT compiling is lower, which is partly due to the new implementation of the `nearest_neighbors` subtask not needing compilation, since it is handled by calling a precompiled C++ implementation. Additionally the `fuzzy_simplicial_set` subtask no longer is JIT-compiled, since in GPUMAP it consists mainly of calls to `cupy` and GPU-compiled methods.

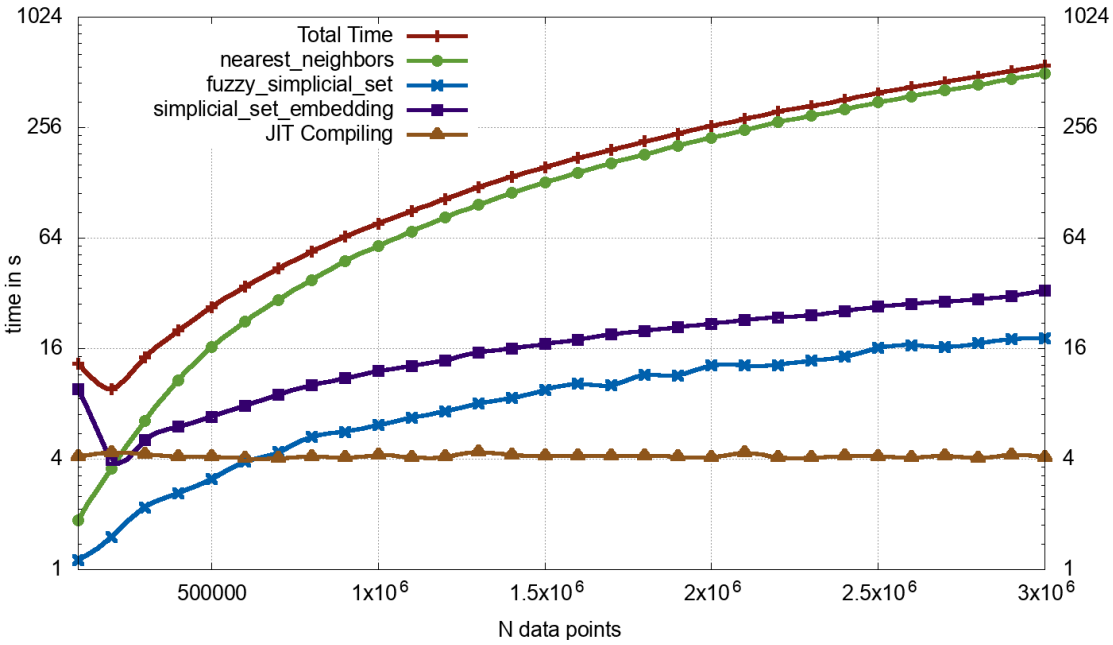


Figure 5.1: Times taken by GPUMAP subtasks on reduced GoogleNews data sets.

In Table 5.2 GPUMAP is compared with t-SNE-CUDA. A comparison on the same hardware was not possible, as no working version of t-SNE-CUDA could be obtained. Neither the recommended installation method via `conda`, nor a manual compilation version, resulted in working installations, despite closely following the recommended steps of the installation description.[1] The code repository has issues reported for the encountered errors.

Therefore, the times used for the comparison are from the t-SNE-CUDA publication [6]. Since they were measured on an Nvidia® Titan X GPU, the performance of t-SNE-CUDA and GPUMAP cannot directly be compared based on the differences. However, they still capture how the algorithms scale in comparison. The data sets used are MNIST, CIFAR and the GloVe data set [41]. Since the times of [6] do not include the classification parts of CIFAR and MNIST, the data set sizes are smaller, which is why the times in Table 5.2 differ from those of Table 5.1.

[1] `https://github.com/CannyLab/tsne-cuda/wiki/Installation`, accessed 28.04.2019

Table 5.2: Comparison of GPUMAP and t-SNE-CUDA on the CIFAR, MNIST and GloVe data sets.

Data set	N	M	t-SNE-CUDA	GPUMAP	JIT	non-JIT	KNN
MNIST	60 K	784	6.98	7.74	4.12	3.62	1.40
CIFAR	50 K	3072	12[2]	8.49	4.02	4.47	2.34
GloVe	2.2 M	300	573	322	4.14	318	284

Table 5.2 also features columns for the time taken for JIT compilation and the remaining time. Since t-SNE-CUDA is compiled beforehand, comparing the two algorithms by actual execution time should be done based on the non-JIT time. Further the KNN column shows the time spent on the `nearest_neighbors` subtask. Both implementations have such a subtask, that both solve by using the FAISS library. But while GPUMAP spends ~88% of time on its total time for processing the GloVe data set, the t-SNE-CUDA publication only lists an approximate percentage between 12%-15% spent on the subtask. In absolute numbers this would mean a time of 69 to 86 seconds spend for the KNN search, which is less than one third of the time GPUMAP takes.

5.2 Visualization Quality

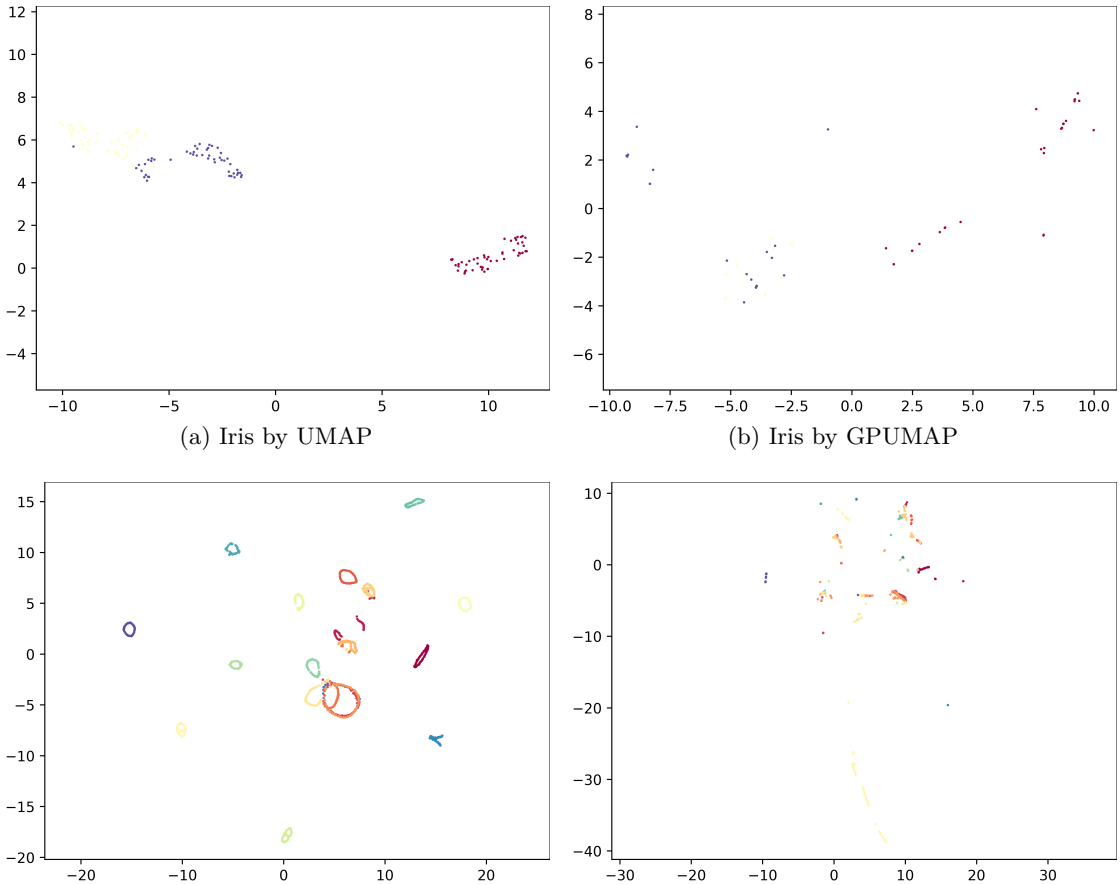
As the most common use case of UMAP is to perform dimensionality reduction for visualizations, GPUMAP needs to provide visualizations of similar quality to be viable. Figure 5.2 and Figure 5.3 show comparisons of UMAP on the left and GPUMAP on the right.

As can be seen the visualizations produced by GPUMAP contain more "noise". Points are not as closely clustered together as the UMAP equivalents. Especially for small data sets this is visible. The points of the small Iris data set are spread out, even though GPUMAP already uses twice as many epochs for the creation of the embedding than UMAP.

For the COIL-20 data set, the circular shapes formed by UMAP are not reproduced by GPUMAP. While the points are still grouped together in the same way as UMAP, they form clumps or trails of points instead of circles. Further a group of points was located to the bottom of the visualization, which causes the visualization to look more distorted and even less similar looking to the visualization of UMAP.

For bigger data sets the observed "noise" carries less weight as for small data sets. The produced visualizations of GPUMAP are more or less aligned with those of UMAP. The clusters of MNIST and F-MNIST are clearly recognizable and closely resemble those produced by UMAP.

[2] "we can run on the full [CIFAR] set in under 12 seconds", stated in [6], p. 5.



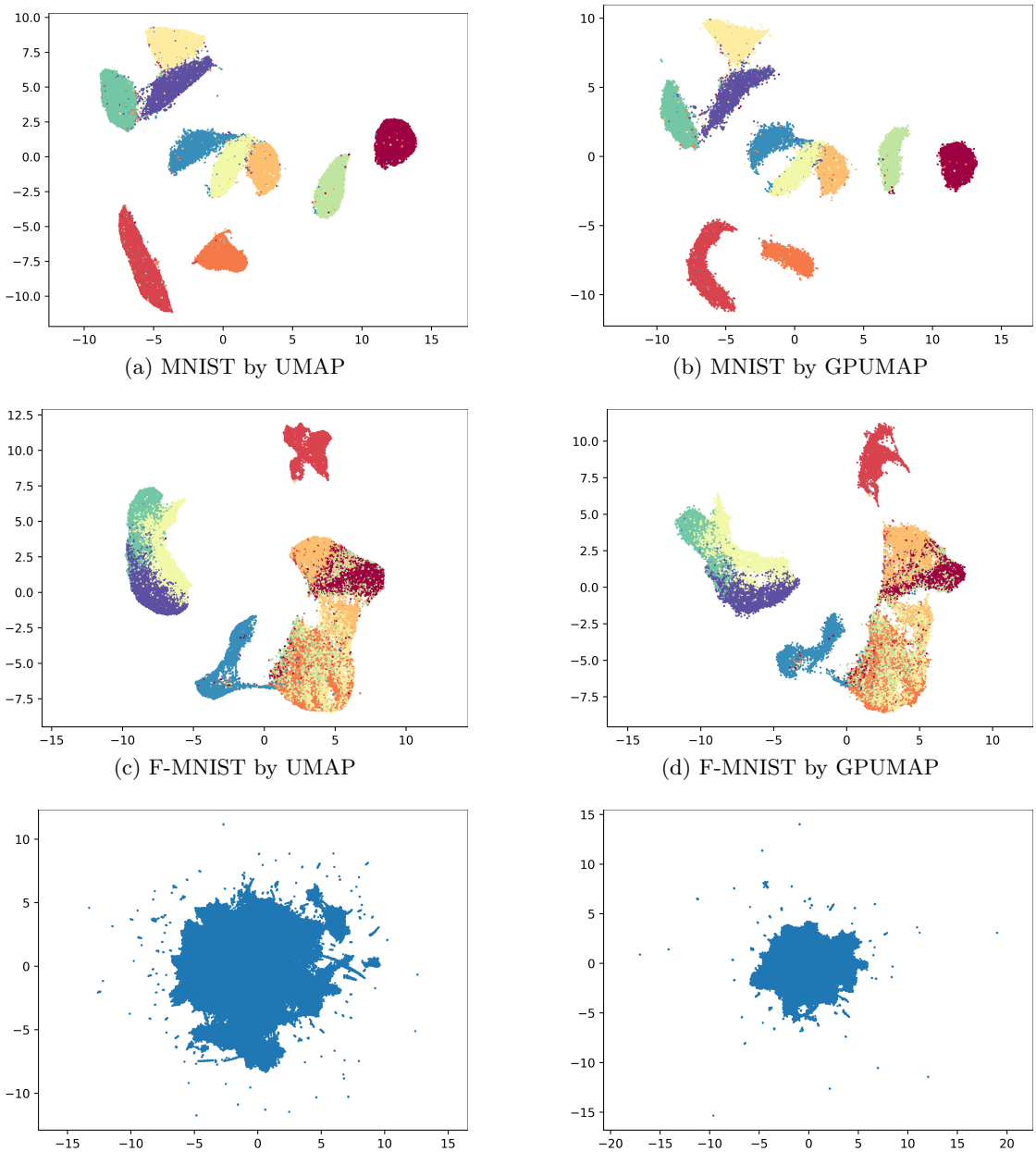
(a) Iris by UMAP

(b) Iris by GPUMAP

(c) COIL-20 by UMAP

(d) COIL-20 by GPUMAP

Figure 5.2: Comparison of visualizations by GPUMAP and UMAP on Iris and COIL-20.



(a) MNIST by UMAP

(b) MNIST by GPUMAP

(c) F-MNIST by UMAP

(d) F-MNIST by GPUMAP

(e) GoogleNews by UMAP

(f) GoogleNews by GPUMAP

Figure 5.3: Comparison of visualizations by GPUMAP and UMAP on MNIST, F-MNIST and GoogleNews.

6 Discussion

6.1 Advantages of GPUMAP

First and foremost, GPUMAP is noticeably faster than UMAP. The achieved speedups of up to 20-60 times would not have been possible with CPUs, since even when assuming an ideal linear speedup, CPUs commonly do not have 20 units that can compute in parallel. Considering the hardware development trend, the performance gap between GPUs and CPUs will keep growing, allowing for additional speedup of GPUMAP over CPU versions with upcoming hardware.

By mostly running on the GPU, GPUMAP also has the benefit of not competing for CPU resources. This is an advantage e.g. if other computations are performed on the CPU and results of running computations have to be visualized.

Further it can be argued by means of the estimations made in Table 5.2, that GPUMAP performs as good or better than other state of the art GPU-parallel visualization algorithms. Especially when neglecting the time spent on JIT compilation, GPUMAP outperforms the t-SNE-CUDA implementation.

6.2 Drawbacks of GPUMAP

GPUMAP unfortunately also has several drawbacks. While it supports all features of the sequential version, not all are available as a GPU-parallel implementation. Apart from allowing to embed into already existing embeddings, GPUMAP neither supports sparse matrices nor user-defined distance metrics.

Further, the performance gains by the GPU parallelization are not drastic enough in its present state. While all subtasks that were implemented in parallel show good improvements, the usage of the FAISS library did not result in improvements as high as expected. Other implementations, i.e. t-SNE-CUDA, also use the library with better results. Most likely this is an implementation error, but since multiple versions have been tried out, including one equal to the t-SNE-CUDA implementation, it should also be considered to replace the library with another KNN search algorithm.

Finally, the visualizations of GPUMAP contain a lot of “noise”. Raising the number of iterations in which the low-dimensional representation is getting adapted, partly solves this problem for big data sets. However, the visualization of small data sets or such with fine structures as seen in Figure 5.2 are of too poor quality, to recommend using GPUMAP for visualizing such data sets.

The noise could be caused by outdated values, since the threads running in parallel on the GPU all work on the same data created in the previous iteration. They don’t see the updates that the other threads make until after the iteration. This could cause points to be moved too far or too little, so that after the iteration some points that should have been moved closer together, end up getting moved past each other. Or points that should have been moved slightly apart, end up getting distanced too far from each other.

7 Conclusion

7.1 Summary

The new implementation of UMAP for GPUs, named GPUMAP, is a fast dimensionality reduction algorithm. It can compete with the currently fastest implementation of the status quo algorithm, t-SNE-CUDA. While further improvements and features need to be added for GPUMAP to be a complete drop-in replacement for its sequential counterpart UMAP, it can already be used to create decent visualizations for big data sets in short time. Installation instructions and usage examples can be found in the online repository of GPUMAP under `https://github.com/p3732/gpumap`.

7.2 Future Works

Several improvements can still be made to GPUMAP. Most importantly the visualization quality can be bettered, to more closely resemble that of UMAP. For this, an improvement to the last algorithm subtask, the optimization of the embedding, is required. Since the problem is presumed to stem from outdated values, a sequential processing of the last few iterations could be investigated.

Further, as observed in Section 5.1, the library used to perform the KNN search does not perform as well for GPUMAP as for other projects. While further optimization of how the library is used, could improve the performance, a more drastic solution would be to implement a GPU-parallel version of the algorithm used by UMAP, the NN-Descent algorithm [42]. This would provide multiple advantages. Firstly, NN-Descent has a better performance, as shown in the benchmarks of [39]. Secondly, a JIT-compiled implementation with Numba would allow supporting different distance metrics, bringing GPUMAP closer to feature parity with UMAP. Finally, an implementation with Numba would make GPUMAP platform independent, since the currently used FAISS library is not available on the Windows platform.

An additional important feature currently missing from GPUMAP is support for sparse matrices. Data sets that are big in size and dimensionality, but actually only contain comparatively few data points due to sparseness, are common when handling word vectors. Supporting them would allow bigger, sparse, data sets to be visualized quickly.

Apart from processing bigger data sets, another goal is to process small to medium-sized data sets swiftly. If data sets can be visualized fast enough, they can be visually analyzed in close to real-time. While GPUMAP already enables comparatively fast processing, the time spent on JIT-compilation is a constant factor. If the CUDA functions were compiled "ahead of time" (AOT), data sets could be processed even faster. Applications for this are numerable, from neural networks in machine learning, complex statistics in finance to DNA sequencing in biology.

7.3 Closing Words

This thesis and the scripts with which the profilings and visualizations were produced, are available under `https://github.com/p3732/master_thesis`.

Bibliography

[1] L. Van Der Maaten and G. Hinton, "Visualizing High-Dimensional Data Using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008. Available: http://www.jmlr.org/papers/v9/vandermaaten08a.html. [Accessed: 18-Mar-2019]

[2] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *ArXiv e-prints*, 2018. Available: http://arxiv.org/abs/1802.03426

[3] N. Pezzotti, A. Mordvintsev, T. Hollt, B. P. F. Lelieveldt, E. Eisemann, and A. Vilanova, "Linear tSNE Optimization for the Web," *ArXiv preprint*, 2018. Available: http://arxiv.org/abs/1805.10817

[4] Mozilla Foundation, "WebGL.". Available: https://www.khronos.org/webgl/. [Accessed: 17-Apr-2019]

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," in *Proceedings of the IEEE*, 1998, vol. 86, pp. 2278–2324. Available: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf. [Accessed: 17-Apr-2019]

[6] D. M. Chan, R. Rao, F. Huang, and J. F. Canny, "t-SNE-CUDA: GPU-Accelerated t-SNE and its Applications to Modern Data," *ArXiv preprint*, 2018. Available: http://arxiv.org/abs/1807.11824

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[8] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *ArXiv preprint*, 2017. Available: http://arxiv.org/abs/1702.08734

[9] NVIDIA Corporation, "CUBLAS Library." Available: https://docs.nvidia.com/cuda/cublas/. [Accessed: 17-Apr-2019]

[10] J. Barnes and P. Hut, "A Hierarchical O(N log N) Force-Calculation Algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[11] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm," *GPU Computing Gems Emerald Edition*, pp. 75–92, 2011.

[12] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] D. Ulyanov, "Multicore-TSNE," *GitHub repository.* https://github.com/DmitryUlyanov/Multicore-TSNE; GitHub, 2016.

[14] J. Heer, M. Bostock, and V. Ogievetsky, "A Tour through the Visualization Zoo," *Communications of the ACM*, vol. 53, no. 5, pp. 59–67, 2010. Available: http://arxiv.org/abs/arXiv:1011.1669v3

[15] S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[16] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* Cambridge University Press, 2012.

[17] D. Arthur and S. Vassilvitskii, "k-means++: The Advantages of Careful Seeding," in *Proceedings of the eighteenth annual acm-siam symposium on discrete algorithms*, 2007, pp. 1027–1035. Available: http://portal.acm.org/citation.cfm?id=1283494. [Accessed: 17-Apr-2019]

[18] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, "NP-hardness of Euclidean sum-of-squares clustering," *Machine Learning*, vol. 75, no. 2, pp. 245–248, 2009.

[19] C. M. Bishop, *Pattern Recognition and Machine Learning*, vol. 16. 2007.

[20] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the second international conference on knowledge discovery and data mining*, 1996, pp. 226–231.

[21] J. Gan and Y. Tao, "DBSCAN Revisited: Mis-Claim, Un-Fixablility, and Approxmimation," *Sigmod*, pp. 519–530, 2015. Available: https://dl.acm.org/citation.cfm?id=2737792. [Accessed: 17-Apr-2019]

[22] M. Götz, C. Bodenstein, and M. Riedel, "HPDBSCAN: Highly parallel dbscan," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015, pp. 2:1–2:10.

[23] L. Van Der Maaten, "Accelerating t-SNE using Tree-Based Algorithms," *Journal of Machine Learning Research*, vol. 15, no. 2014, pp. 3221–3245, 2014. Available: http://jmlr.org/papers/v15/vandermaaten14a.html. [Accessed: 17-Apr-2019]

[24] I. T. Jolliffe, "Principal Component Analysis, Second Edition," *Springer Series in Statistics*, vol. 98, p. 487, 2002.

[25] M. Andrecut, "Parallel GPU Implementation of Iterative PCA Algorithms," *ArXiv preprint*, 2008. Available: http://arxiv.org/abs/0811.1081

[26] G. E. Hinton and S. T. Roweis, "Stochastic Neighbor Embedding," *Advances in Neural Information Processing Systems*, vol. 15, pp. 833–840, 2002. Available: http:

//papers.nips.cc/paper/2276-stochastic-neighbor-embedding.pdf. [Accessed: 17-Apr-2019]

[27] S. A. Nene, S. K. Nayar, and H. Murase, "Columbia Object Image Library (COIL-20)," Columbia University, 1996. Available: http://www.cs.columbia.edu/CAVE/software/softlib/coil-20.php. [Accessed: 17-Apr-2019]

[28] P. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1993, pp. 311–321. Available: http://web.cs.iastate.edu/~honavar/nndatastructures.pdf. [Accessed: 17-Apr-2019]

[29] L. McInnes, "PyData Ann Arbor: Leland McInnes | PCA, t-SNE, and UMAP: Modern Approaches to Dimension Reduction [Video file]." 2018. Available: https://www.youtube.com/watch?v=YPJQydzTLwQ

[30] W. Dong, M. Charikar, and K. Li, "Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures," in *Proceedings of the 20th international conference on world wide web*, 2011, pp. 577–586.

[31] R. A. Fisher, "The Use of Multiple Measurements in Taxonomic Problems," *Annals of Eugenics*, vol. 7, no. 7, pp. 179–188, 1936.

[32] E. Alpaydin and F. Alimoglu, "Pen-Based Recognition of Handwritten Digits Data Set." University of California, Irvine. Machine Learning Repository, 1998.

[33] S. A. Nene, S. K. Nayar, and H. Murase, "Columbia Object Image Library (COIL-100)," Columbia University, 1996. Available: http://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php. [Accessed: 19-Apr-2019]

[34] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments," University of Massachusetts, Amherst, 07-49, 2007. Available: http://vis-www.cs.umass.edu/lfw/. [Accessed: 23-Apr-2019]

[35] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," 28-Aug-2017.. Available: http://arxiv.org/abs/cs.LG/1708.07747

[36] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed: 21-Apr-2019]

[37] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, Oct. 2013.

[38] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT Compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 7:1–7:6. Available: http://doi.acm.org/10.1145/2833157.2833162

[39] M. Aumüller, E. Bernhardsson, and A. Faithfull, "ANN-Benchmarks: A Benchmarking Toolfor Approximate Nearest Neighbor Algorithms," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10609 LNCS, no. 614331, pp. 34–49, 2017. Available: http://ann-benchmarks.com/. [Accessed: 24-Apr-2019]

[40] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf. [Accessed: 27-May-2019]

[41] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. Available: http://www.aclweb.org/anthology/D14-1162. [Accessed: 28-May-2019]

[42] W. Dong, C. Moses, and K. Li, "Efficient K-nearest Neighbor Graph Construction for Generic Similarity Measures," in *Proceedings of the 20th International Conference on World Wide Web*, 2011, pp. 577–586. Available: http://doi.acm.org/10.1145/1963405.1963487. [Accessed: 28-Apr-2019]