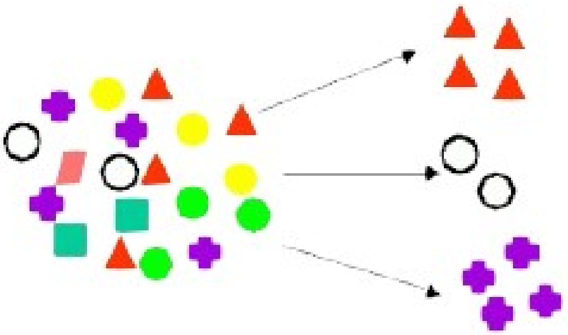


Efficient Evidence Accumulation Clustering for large datasets/big data

Diogo Alexandre Oliveira Silva

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Ana Fred 1 and Helena Aidos 2

Examination Committee

Chairperson:	Professor Full Name
Supervisor:	Professor Full Name 1 (or 2)
Member of the Committee:	Professor Full Name 3

Month 2015

Dedicated to someone special...

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: palavra-chave1, palavra-chave2,...

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: keyword1, keyword2,...

Contents

List of Tables

List of Figures

Glossary

API	Application Programming Interface.
CPU	Central Processing Unit.
EAC	Evidence Accumulation Clustering.
GPGPU	General Purpose computing in Graphics Processing Units.
GPU	Graphics Processing Unit.
HAC	Hierarchical Agglomeration Clustering.
PCA	Principal Component Analysis.
PC	Principal Component.
QK-Means	Quantum K-Means.
Qubit	Quantum bit.
SL-HAC	Single-Linkage Hierarchical Agglomeration Clustering.
SVD	Singular Value Decomposition.

Chapter 1

Introduction

1.1 Challenges and Motivation

Advances in technology allow for the collection and storage of unprecedented amount and variety of data, a concept commonly designated by *Big Data*. Most of this data is stored electronically and there is an interest in automated analysis for generation of knowledge and new insights. The applications of such analysis are abundant and across many fields, ranging from recommender systems and customer segmentation in business, to predicting when a jet engine is likely to fail using sensor data, or even the study of gene expression in biomedics, to namy a few.

A growing body of statistical methods aiming to model, structure and/or classify data already exist, e.g. linear regression, principal component analysis, cluster analysis, support vector machines, neural networks. Cluster analysis is an interesting tool because it typically doesn't make assumptions on the structure of the data. Since, often, the structure of the data is unknown, clustering techniques become particularly interesting for transforming this data into knowledge and discovering its underlying structure and patterns. Clustering is a hard problem and a vast body of work on these algorithms exist. Yet, typically, no single algorithm is able to respond to the specificities of all data. Different methods are suited to datasets of different characteristics and, often, the challenge of the researcher is to find the right algorithm for the task.

Currently, there are state of the art algorithms that are more robust than ”traditional” algorithms by having a wider applicability or being less dependent on input parameters, e.g. algorithms that don't take any parameters for performing an analysis. One such algorithm is Evidence Accumulation Clustering (EAC), belonging to the wider class of ensemble methods. EAC is a state-of-the art clustering algorithm that addresses the robustness challenge. However, the current reality of capturing massive amounts of data rises new challenges. Two important challenges are efficiency and scalability, which translate on how fast the algorithms are and how well they scale when the input data multiplies in size, dimensionality and variety. The algorithms themselves are no longer the only focus of research. Much effort is being put into the scalability and performance of algorithms, which usually translates in addressing their computational complexity with parallelized computation and distributed memory being some of the proposed solutions. Cluster analysis with EAC should be fast and able to scale to larger datasets as well as robust, so as to address the reality of big data.

This dissertation is concerned with pushing the current limits of the EAC to large datasets by addressing the problems of scalability and efficiency without compromising robustness, using technology available in a desktop workstation. Processing of huge amounts of data has been out of the range of capability of the traditional desktop workstation. This sprouted the rise of new uses of existing computing architectures (e.g. Graphic Processing Units) and development of new programming models (e.g. Hadoop, shared and distributed memory). The problem at hand is, then, to optimize the algorithm regarding both speed and memory usage. This, of course, comes with challenges. How can one keep the original accuracy while significantly increase efficiency? Is there an exploitable trade-off between the three main characteristics: speed, memory and accuracy? These are guiding questions that this dissertation addresses.

1.2 Goals

This dissertation aims to research and extend the state of the art of ensemble clustering, in what concerns the EAC method and its application to large datasets, while also assessing algorithmic solutions and parallelization techniques. The goal is to understand EAC's suitability for large datasets and find ways to respond to the stated challenges, in terms of speed and memory. The main objectives for this work are:

- Study the integration of quantum inspired methods in EAC.
- Study the integration of the General Purpose computing in a Graphics Processing Unit (GPGPU) paradigm in EAC.
- Devise strategies to reduce computation and memory complexities of EAC.
- Application of Evidence Accumulation Clustering to Big Data.
- Validation of Big Data EAC on real data.

1.3 Contributions

The main contributions are the adaptation of the three distinct stages of the EAC framework to larger datasets. In particular, an efficient parallel version for Graphics Processing Units (GPU) of the K-Means clustering algorithm is implemented for the first stage of EAC. Still in this stage, two clustering algorithms in the young field of Quantum Clustering were reviewed, tested and evaluated having EAC in mind. Different methods for the second stage were tested, using complete matrices and sparse matrices.Worthy of mention is a novel and specialized method for building a sparse matrix in the second stage. A GPU parallel version of a MST (Minimum Spanning Tree) solver algorithm was reviewed and tested for the last stage, a co-product of which was an algorithm to find the connected components of a MST. A hard disk solution was implemented for dealing with large datasets whose space complexity in the final stage exceeded the available memory.

1.4 Outline

Chapter 2 provides an introduction to clustering nomenclature and concepts, as well as some ”traditional” clustering algorithms. Chapter 3 starts by reviewing the Evidence Accumulation Clustering algorithm in detail. It goes on to review possible approaches to the problem of scaling EAC. Based on an algorithmic approach, a review of the young field of quantum clustering is presented, with a more in-depth emphasis on two algorithms. With a parallelization approach in mind, a programming model for the GPU (CUDA) is reviewed, followed by some parallelized versions of relevant algorithms to the problem of this dissertation. The following chapter, **??**, presents the approach that was actually taken to scale EAC. It presents the steps taken on each part of the algorithm, the underlying difficulties and what was done to address them. It also includes the reference of approaches that were developed but were not deemed suited to integrate the EAC toolchain. In the results chapter, **??**, the results of the different approaches of optimizing the EAC method are presented. Chapter **??** presents an interpretation and critical discussion of those results. Finally, chapter **??** concludes the dissertation. It also offers recommendations for future work.

Chapter 2

Clustering: basic concepts, definitions and algorithms

Hundreds of methods for data analysis exist. Many of these methods fall into the realm of machine learning, which is usually divided into 2 major groups: *supervised* and *unsupervised* learning. Supervised learning deals with labeled data, i.e. data for which ground truth is known, and tries to solve the problem of classification. Examples of supervised learning algorithms are Neural Networks, Decision Trees, Linear Regression and Support Vector Machines. Unsupervised learning deals with unlabeled data for which no extra information is known. An example of algorithms within this paradigm is clustering algorithms, which are the focus of this chapter.

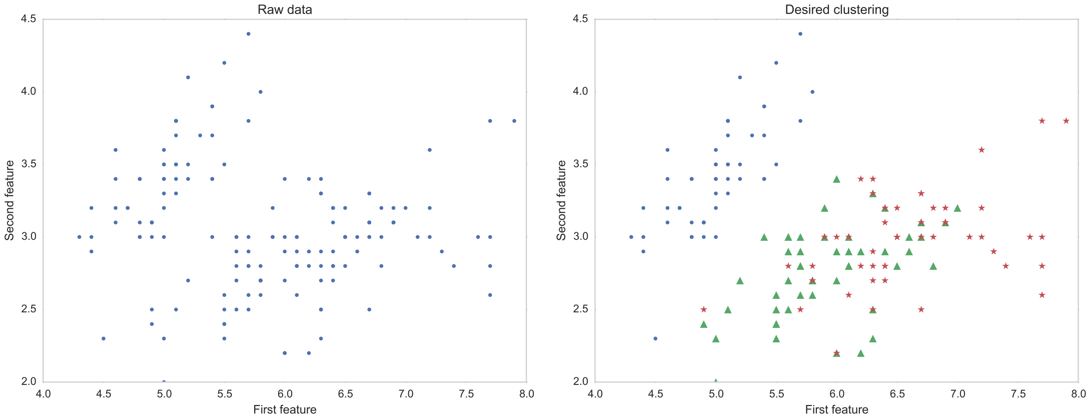
This chapter will serve as an introduction to clustering. It starts by defining the problem of clustering in section 2.1, goes on to provide useful definitions and notation in section 2.2 and briefly addresses different properties of clustering algorithms in section 2.3. Two very well known algorithms are presented: K-Means in section 2.4 and Single-Link in section 2.5. Evidence Accumulation Clustering is a state of the art ensemble clustering algorithm and the focus of this dissertation. Section 2.6 will explain briefly the concept of ensemble clustering followed by an overview and application examples of the EAC algorithm in section 2.7.

2.1 The problem of clustering

Cluster analysis methods are unsupervised and the backbone of the present work. The goal of data clustering, as defined by [2], is the discovery of the *natural grouping(s)* of a set of patterns, points or objects. In other words, the goal of data clustering is to discover structure on data. The methodology used is to group patterns (usually represented as a vector of measurements or a point in space [3]) based on some similarity, such that patterns belonging to the same cluster are typically more similar to each other than to patterns of other clusters. Clustering is a strictly data-driven method, in contrast with classification techniques which have a training set with the desired labels for a limited collection of patterns. Because there is very little information, as few assumptions as possible should be made about the structure of the data (e.g. number of clusters). And, because clustering typically makes as few assumptions on the data as possible, it is appropriate to use it on exploratory structural analysis of the data. The process of clustering data has three main stages [3]:

- **Pattern representation** refers to the choice of representation of the input data in terms of size, scale and type of features. The input patterns may be fed directly to the algorithms or undergo *feature selection* and/or *feature extraction*. The former is simply the selection of which features of the originally available should be used. The later deals with the transformation of the original features such that the resulting features will produce more accurate and insightful clusterings, e.g. Principal Component Analysis.
- **Pattern similarity** refers to the definition of a measure for computing the similarity between two patterns.
- **Grouping** refers to the algorithm that will perform the actual clustering on the dataset with the defined pattern representation, using the appropriate similarity measure.

As an example, Figure 2.1a shows the plot of the Iris data set [4, 5], a small well-known Machine Learning data set. This data set has 4 features, of which only 2 are represented, and 3 classes, of which 2 are overlapping. A class is overlapping another if they share part of the feature space, i.e. there is a zone in the feature space whose patterns might belong to either class. Figure 2.1b presents the desired clustering for this data set.



(a) Input data, unlabeled.

(b) Desired labels.

Figure 2.1: First and second features of the Iris dataset. Fig. 2.1a shows the raw input data, i.e. how the algorithms ”see” the data. Fig. 2.1b shows the desired labels for each point, where each color is coded to a class.

2.2 Definitions and Notation

This section will introduce relevant definitions and notation within the clustering context that will be used throughout the rest of this document and were largely adopted from [3].

A *pattern* $\mathbf{x}$ is a single data item and, without loss of generalization, can be represented as a vector of d *features* x_i that characterize that data item, $\mathbf{x} = (x_1, \ldots, x_d)$, where d is referred to as the dimensionality of the pattern. A *pattern set* (or data set) $\mathcal{X}$ is then the collection of all n patterns $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. The number of features is usually the same for all patterns in a given pattern set.

In cluster analysis, the desired clustering, typically, is one that reflects the natural structure of the data, i.e. the original ground truth labeling. In other words, one wants to group the patterns that came from the same state of nature when they were generated, the same *class*. A class, then, can be viewed as a source of patterns and the effort of the clustering algorithm is to group patterns from the same source. Throughout this work, these classes will also be referred to as the ”natural” or ”true” clusterings. *Hard* clustering (or partitional) techniques assign a class label l_i to each pattern $\mathbf{x}_i$. The whole set of labels corresponding to a pattern set $\mathcal{X}$ is given by $\mathcal{L} = \{l_1, \ldots, l_n\}$, where l_i is the label of pattern $\mathbf{x}_i$. Closely related to the whole set of labels is the concept of a *partition*, which completely describes a clustering. A partition P is a collection of k *clusters*. A cluster C is a subset of nc patterns $\mathbf{x}_i$ taken from the pattern set, where the patterns belonging to one subset don't belong to any other in the same partition. A clustering *ensemble* $\mathbb{P}$ is a set of partitions from a given pattern set. The relationship between the above concepts is condensed in the following expressions:

$$ensemble \qquad \mathbb{P} = \left\{P^1, P^2, \ldots P^N\right\} \tag{2.1}$$

$$partitions \qquad P^j = \left\{C_1^j, C_2^j, \ldots C_{k_j}^j\right\} \tag{2.2}$$

$$clusters \qquad C_i^j = \left\{x_1, x_2, \ldots x_{nc_i^j}\right\} \tag{2.3}$$

Finally, a *similarity measure* is a way of quantifying how similar two patterns are in the feature space. In contrast, a *dissimilarity measure* measures how dissimilar two patterns are. Examples of this measures are a metric (such as the euclidean distance) and the Pearson correlation coefficient.

2.3 Characteristics of clustering techniques

Clustering algorithms may be categorized and described according to different properties. For the sake of completeness, a brief discussion of some of their properties will be layed out in this section.

It's common to organize cluster algorithms into two distinct types: *partitional* and *hierarchical*. A partitional algorithm, such as K-Means, is a hard clustering algorithm that will output a partiton where each pattern belongs exclusively to one cluster. A hierarchical algorithm produces a tree-based data structure called *dendrogram*. A dendrogram contains different partitions at different levels of the tree which means that the user can easily change the desired number of clusters by simply traversing the different levels. This is an advantage over a partitional algorithm since a user can analyze different partitions with different numbers of clusters without having to rerun the algorithm. Hierarchical algorithms can be further split into two approaches: bottom-up (or *agglomerative*) and top-down (or *divisive*). The former starts with all patterns as distinct clusters and will group them together according to some dissimilarity measure, building the dendrogram from the ground up; examples of algorithms that take this approach are Single-Link and Average-Link. The later will start will all patterns in the same cluster and continuosly split it until all patterns are separated, building the dendrogram from the top level to the bottom; this approach is taken by the Principal Directon Divisive Partitioning[6] and Bisecting K-Means [7] algorithms.

Another characteristic relates to how algorithms use the features for computing similarities. If all features are used simultaniously the algorithm is called *polithetic*, e.g. K-Means. Otherwise, if the features are used sequentially, it's called *monothetic*, e.g. [8].

Contrasting *hard* clustering algorithm are the *fuzzy* algorithms. A fuzzy algorithm will attribute to each pattern a degree of membership to each cluster. A partition can still be extracted from this output by choosing, for each pattern, the cluster with higher degree of membership. An example of a fuzzy algorithm is the Fuzzy C-Means [9].

Another characteristic is an algorithm's stochasticity. A *stochastic* algorithms uses a probabilistic process at some point in the algorithms, possibly yielding different results in each run, e.g. K-Means can use a random initialization. As an example, the K-Means algorithm typically picks the initialization centroids randomly. A *deterministic* algorithm, on the other hand, will always produce the same result for a given input, e.g. Single-Link.

Finally, the last characteristic discussed is how an algorithm processes the input data. An algorithm is said to be *incremental* if it processes the input incrementally, i.e. taking part of the data, processing it and then doing the save for the remaining parts, e.g. PEGASUS [10]. A *non-incremental* algorithm, on the other hand, will process the whole input in each run, e.g. K-Means. This discussion is specially relevant when considering large datasets that may not fit in memory or whose processing would take too long for a single run and is therefore done in parallel.

2.4 K-Means

One of the most famous non-optimal solutions for the problem of partitional clustering is the K-Means algorithm [11]. The K-Means algorithm uses K *centroid* representatives, c_k, for K clusters. Patterns are assigned to a cluster such that the squared error (or, more accurately, squared dissimilarity measure) between the cluster representatives and the patterns is minimized. In essence, K-Means is a solution (although not necessarily an optimal one) to an optimization problem having the Sum of Squared Errors as its objective function, which is known to be a computationally NP hard problem [2]. It can be mathematically demonstrated that the optimal representatives for the clusters are the means of the patterns of each cluster [12]. K-Means, then, minimizes the following expression, where the distance measure used is the Euclidean distance:

$$\sum_{k=1}^{K} \sum_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - c_k\|^2 \tag{2.4}$$

K-Means needs two initialization parameters: the number of clusters and the centroid initializations. It starts by assigning each pattern to its closer cluster based on the cluster's centroid. This is called the **labeling** step since one usually uses cluster labels for this assignment. The centroids are then recomputed based on this assignment, in the **update** step. The new centroids are the mean of all the patterns belonging to the clusters, hence the name of the algorithm. These two steps are executed iteratively until a stopping condition is met, usually the number of iterations, a convergence criteria or both. The initial centroids are usually chosen randomly, but other schemes exist to improve the overall accuracy of the algorithm, e.g. K-Means++ [13]. There are also methods to automatically choose the number of clusters [12].

The dissimilarity measure used is typically the Euclidean distance. This tends to produce hyper-spherical clusters [3]. Still, according to [2], other measures have been used such as the L1 norm, Mahalanobis distance, as well as the cosine dissimilarity [12]. The choice of similarity measure must be made carefully as it may not guarantee that the algorithm will converge.

A detail of implementation is what to do with clusters that have no patterns assigned to them. One approach to this situation is to drop the empty clusters in further iterations. However, allowing the existence of empty clusters or dropping empty clusters is undesirable since the number of clusters is an input parameter and it's expected that the output contains the specified number of clusters. Other approaches exist dealing with this problem, such as equaling the centroid of an empty cluster to the pattern furthest away from its assigned centroid or reusing the old centroids as in [14].

K-Means is a simple algorithm with reduced complexity $O(ntk)$, where k is the number of clusters and t is the number of iterations that it executes. Because of this, K-Means is often used as foundational step of more complex and robust algorithms, such as the EAC algorithm.

As an example, the evolution and output of the K-means algorithm to the data presented in Fig. 2.1 is represented in Fig. 2.2. The algorithm was executed with 3 random centroids.

Even with the correct number of clusters, the clustering results don't match 100% the natural clusters. The accuracy relative to the natural clusters of Fig. 2.1b is 88% as measured by the Consistency Index (CI) [15]. In this example, the problem is the two overlapping clusters. It's hard for an algorithm to discriminate between two clusters when they have similar patterns. When no prior information about the dataset is given, the number of clusters can be hard to discover. This is why, when available, a domain expert may provide valuable insight on tuning the initialization parameters.

2.5 Single-Link

Single-Link [16] is one of the most popular hierarchical agglomerative clustering (HAC) algorithms. HAC algorithms operate over a pair-wise dissimilarity matrix and outputs a dendrogram (e.g. Fig 2.4a). The main steps of an agglomerative hierarchical clustering algorithm are the following [3]:

1. Create a pair-wise dissimilarity matrix of all patterns, where each pattern is a distinct cluster;

2. Find the closest clusters, merge them and uptade the matrix to reflect this change. The rows and

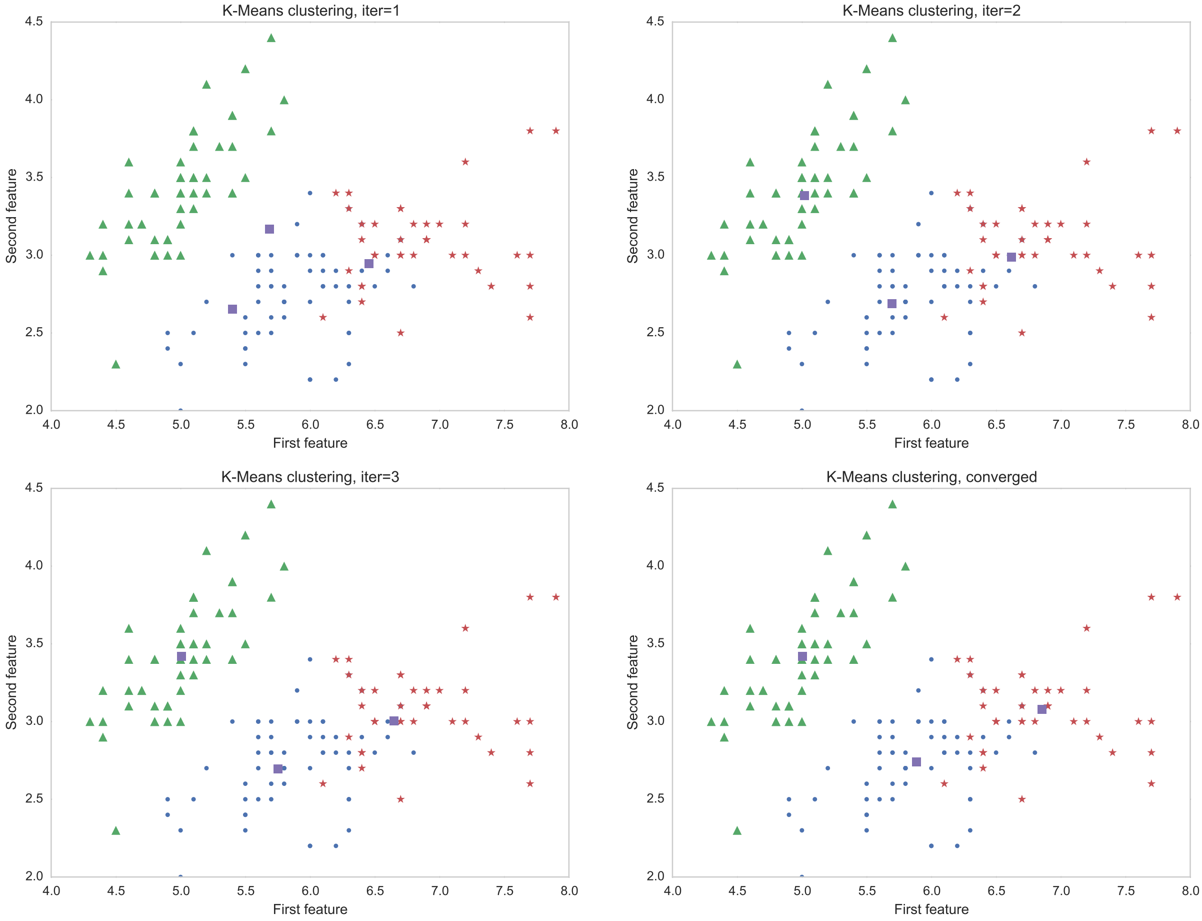


Figure 2.2: The output labels of the K-Means algorithm with the number of clusters (input parameter) set to 3. The different plots show the centroids (squares) evolution on each iteration. Between iteration 3 and the converged state 2 more iterations were executed.

columns of the two merged clusters are deleted and a new row and column are created to store the new cluster.

3. Stop if all patterns belong to a single cluster, otherwise continue to step 2.

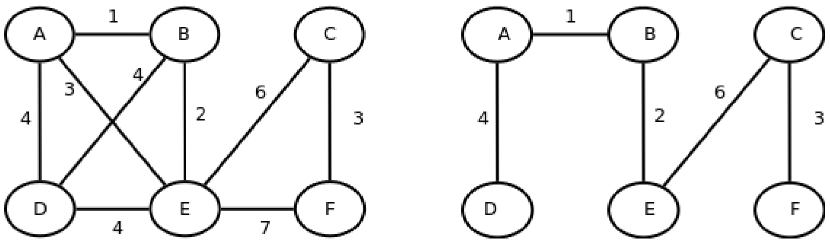
The algorithm stops when $n-1$ merges have been performed, which is when all patterns have been connected in the same cluster. Just like in the K-Means algorithm, different similarity measures can be used for the distances.

The proximity between clusters in the second step is distinguishes between the different HAC linkage algorithms, such as Single-Link , Average-Link, Complete-Link, among others. In Single-Link (SL), the proximity between any two clusters is the the dissimilarity between their closest patterns. On the other hand, in Complete-Link, it's the proximity between the their most distant patterns and, in Average-Link, is the proximity between the average point of each cluster. In SL, because the algorithm connects first clusters that are more similar, it naturally gives more importance to regions of higher density [12].

The total time complexity of a naive implementation is $O(n^3)$ since it performs a $O(n^2)$ search in step two and it does it $n-1$ times. Over time, more efficient implementations have been proposed, such as SLINK [17]. SLINK needs no working copy of $O(n^2)$ the pair-wise similarity matrix (if the original can be modified), has a working memory of $O(n^2)$ and time complexity of $O(n^2)$. This increase in performance comes from the observation that the $O(n^2)$ search can be transformed in a $O(n)$ search at the expense of keeping two arrays of length n that will store the most similar cluster for each pattern and the corresponding similarity measure. This way, to find the two closest clusters, the algorithm will not search the entire similarity matrix, but only the new similarity array since this array keeps the closest cluster of each cluster. Naturally these arrays must be updated upon a cluster merge.

An interesting property of the SL algorithm is its equivalence with a Minimum Spanning Tree (MST), an observation first made by [18]. In graph theory, a MST is a tree that connects all vertices together while minimizing the sum of all the distance between them. An example of a graph and its corresponding MST can be seen in Fig. 2.3. In this context, the edges of the MST are the distances between the patterns and the vertices are the patterns themselves. A MST contains all the information necessary to build a Single-Link dendrogram. To walk down through the levels of the dendrogram from the MST, one cuts the least similar edges. Furthermore, this approach can be used to apply Single-Link clustering to graphs-encoded problems in a straight-forward way. Furthermore, the performance properties of this method are roughly the same as SLINK [19].

The true advantage of using an MST based approach comes when the number of edges (similarities) m of the MST is less than $\frac{n(n-1)}{2}$, where n is the number of nodes (patterns) [20]. This is because SLINK works over a inter pattern similarity matrix, meaning that the similarity between every pair of patterns must be explicitly represented. The minimum number of similarities is $\frac{n(n-1)}{2}$, which is equivalent to the upper or lower half triangular matrices of the similarity matrix. The MST, on the other hand, works over a graph that may or may not have edges between every pair of nodes. Fast MST algorithms have a time complexity of $O(mlogn)$, which is an improvement over $O(n^2)$ when $m << \frac{n(n-1)}{2}$.

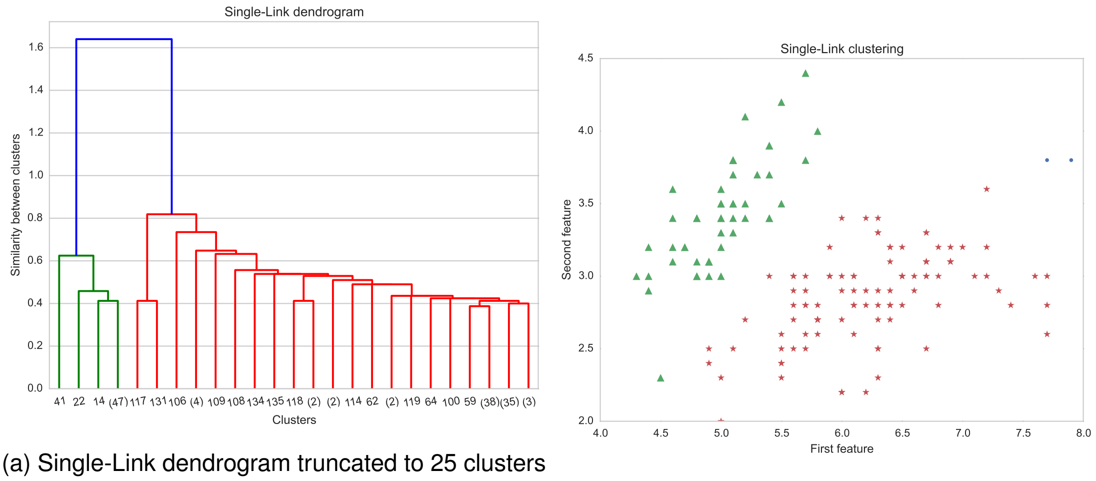


(a) Example of a graph.

(b) MST of the graph to the left.

Figure 2.3: The above figures show an example of a graph (left) and its corresponding Minimum Spanning Tree (right). The circles are vertices and the edges are the lines linking the vertices.

An example of a Single-Link dendrogram and resulting cluster can be observed in Fig. 2.4. The dendrogram in Fig. 2.4a has been truncated to 25 clusters in the bottom level for the sake of readability. The clustering presented on Fig. 2.4b is the result of cutting the dendrogram such that only 3 clusters exist (the number of classes). The accuracy, as measured by the CI, is of 58%.



(a) Single-Link dendrogram truncated to 25 clusters in the bottom level.

(b) Clustering with 3 clusters.

Figure 2.4: The above plots show the dendrogram and a possible clustering taken from a Single-Link run over the Iris data set. Fig. 2.4b was obtained by performing a cut on a level that would yield a partition of 3 clusters.

2.6 Ensemble Clustering

The underlying idea behind ensemble clustering is to take a collection of partitions, a *clustering ensemble*, and combine it into a single partition. There are several motivations for ensemble clustering. Data from real world problems appear in different configurations regarding shape, cardinality, dimensionality, sparsity, etc. Different clustering algorithms are appropriate for different data configurations, e.g. K-Means tends to group patterns in hyperspheres [3] so it is more appropriate for data whose structure is formed by hypershere like clusters. If the true structure of the data at hand is heterogeneous in its configuration, a single clustering algorithm might perform well for some part of the data while other performs better for some other part. Since different algorithms can be used to produce the partitions in the ensemble, one can use a mix of algorithms to address different properties of the data such that the combination is more **robust** to noise and outliers [21] and the final clustering has a **better quality** [12]. Ensemble clustering can also be particularly useful in situations where one doesn't have direct access to all the features of a given data set but can have access to partitions from different subsets and later combining with an ensemble algorithm. Furthermore, the generation of the clustering ensemble can be **parallelized and distributed** since each partition is independent from every other partition.

A clustering ensemble, according to [22], can be produced from (1) different data representations, e.g. choice of preprocessing, feature selection and extraction, sampling; or (2) different partitions of the data, e.g. output of different algorithms, varying the initialization parameters on the same algorithm.

Ensemble clustering algorithms can take three main distinct approaches [12]. The MMCE [21] and BCE [23] are examples of a probabilistic approach. EAC [22] and CSPA [24] are examples of pair-wise similarity based approach, where the algorithms use a co-associations matrix. This approach will be further clarified when the EAC algorithm is explained. HGPA [24], MCLA [24] and bagging [25] are examples of a direct approach to combining the ensemble clusterings, where the algorithms work directly with the labels without creating a co-association matrix. A detailed and thorough review of the similarity measures that can be used on with clustering ensembles and the state of the art algorithms can be consulted in [12].

2.7 Evidence Accumulation Clustering

2.7.1 Overview

The goal of EAC is to find an optimal partition P^* containing k^* clusters, from the clustering ensemble $\mathbb{P}$. The optimal partition should have the following properties [22]:

- **Consistency** with the clustering ensemble;
- **Robustness** to small variations in the ensemble; and,
- **Goodness** of fit with ground truth information, when available.

Ground truth is the true labels of each sample of the dataset, when such exists, and is used for validation purposes. Since EAC is an unsupervised method, this typically will not be available. EAC makes no assumption on the number of clusters in each data partition. Its approach is divided in 3 steps:

1. **Production** of a clustering ensemble $\mathbb{P}$ (the evidence);
2. **Combination** of the ensemble is combined into a co-association matrix;
3. **Recovery** of the natural clusters of the data.

The ensemble of partitions is combined in the second step, where a non-linear transformation turns the ensemble into a co-association matrix, i.e. a matrix $\mathcal{C}$ where each of its elements n_{ij} is the association value between the pattern pair (i, j). The association between any pair of patterns is given by the number of times those two patterns appear clustered together in any cluster of any partition of the ensemble. The rationale is that pairs that are frequently clustered together are more likely to be representative of a true link between the patterns [22], revealing the underlying structure of the data. In other words, a high association n_{ij} means it's more likely that patterns i and j belong to the same class. The construction of the co-association matrix is at the very core of this method.

The co-association matrix itself is not the output of EAC. Instead, it is used as input to other methods to obtain the final partition. The co-association between any two patterns can be interpreted as a similarity measure. Thus, since this matrix is a similarity matrix it's appropriate to use algorithms that take this type of matrices as input, e.g. K-Medoids or hierarchical algorithms such as Single-Link or Average-Link, to name a few. Typically, algorithms use a distance as the similarity, which means that they minimize the values of similarity to obtain the highest similarity between objects. However, a low value on the co-association matrix translates in a low similarity between a pair of objects, which means that the co-association matrix requires prior transformation for accurate clustering results, e.g. replace every similarity value n_{ij} between every pair of object (i, j) by $max\{\mathcal{C}\} - n_{ij}$.

Although any algorithm can be used, the final clustering is usually done using the SL or AL algorithms. Each of this algorithms will take as input the co-association matrix as the similarity matrix. Furthermore, not knowing the ”natural” number of clusters one can use the lifetime criteria, i.e. the number of clusters k should be such that it maximizes the cost of cutting the dendrogram from $k - 1$ clusters to k. Further details on the lifetime strategy for picking the number of clusters falls outside the scope of this work and are presented in [22].

2.7.2 Examples of applications

EAC has been used with success in several areas. Some of its applications are:

- in the field of bioinformatics it was used for the automatic identification of chronic lymphocyt leukemia [26];
- also in bioinformatics it was used for the unsupervised analysis of ECG-based biometric database to highlight natural groups and gain further insight [27];
- in computer vision it was used as a solution to the problem of clustering of contour images (from hardware tools) [28].

Chapter 3

State of the art

Scalability of EAC to large data sets is the concern of this work and, because of that, this chapter starts by reviewing what has been done in terms of scaling EAC in section 3.1 EAC is a method of three parts and this dissertation is concerned with the scalability of the whole algorithm which means that each step must be optimized. Scaling an algorithm means one has to take into account both speed of execution and memory requirements. Increasing speed can be attained with either faster algorithms and/or faster computation of existing algorithms. This chapter reflects research done within both approaches.

Although research on the application of EAC to large data sets hasn't been pursued before, cluster analysis of large data sets has. Since EAC uses traditional clustering algorithms (e.g. K-Means, Single-Link) in its approach, it is useful to understand how scalable the individual algorithms are as they'll have a big impact in the scalability of EAC. Furthermore, valuable insights may be taken from the techniques used in the scalability of other algorithms. To this end, section 3.2 presents a brief review on cluster analysis of large data sets, with a focus on parallelization with GPUs. Furthermore, it offers a more detailed description of a GPU parallel version of K-Means and an approach for parallelizing Single-Link with the GPU.

An alternate approach on clustering for scaling with faster algorithms, the still young field of quantum clustering, was reviewed in section 3.3. This line of research was taken mostly with the first step of EAC in mind.

3.1 Scalability of EAC

The quadratic space and time complexity of processing the $n \times n$ co-association matrix is an obstacle to an efficient scaling of EAC. Two approaches have been proposed to address this obstacle: one dealing with reducing the co-association matrix by considering only the distances of patterns to their p neighbors and the other by using a sparse co-association matrix and maximizing its sparsity.

3.1.1 p neighbors approach

The first approach, [22], proposes an alternative $n \times p$ co-association matrix, where only the p nearest neighbors of each pattern are considered in the evidence combination step. This comes at the cost of having to keep track of the neighbors of each pattern in a separate data structure of $O(np)$ memory complexity and also of pre-computing the p neighbors, which has a time complexity of $O(n^2)$ to compute the similarity measure from each pattern to every other pattern. The quadratic space complexity is then transformed to $O(2np)$: $O(np)$ for the actual co-association matrix and $O(np)$ for keeping track of the neighbors. Since usually one has $p < \frac{n}{2}$ (value for which both this approach and the original $n \times p$ matrix would take the same space), the cost of storing the extra data structure is lower than that of storing an $n \times n$ matrix, e.g. for a dataset with 10^6 patterns and $p = \sqrt{10^6}$ (a value much higher than the 20 neighbors used in [22]), the total memory required for the co-association matrix would decrease from $3725.29GB$ to $7.45GB$ (0.18% of the memory occupied by the complete matrix).

3.1.2 Increased sparsity approach

The second approach, presented in [29], exploits the sparse nature of the co-association matrix. The co-association matrix is symmetric and with a varying degree of sparsity. The former property translates in the ability of storing only the upper triangular of the matrix without any loss on the quality of the results. The later property is further studied with regards to its relationship with the minimum K_{min} and maximum K_{max} number of clusters in the partitions of the input ensemble. The core of this approach is to only store the non-zero values of the upper triangular of the co-association matrix. The authors study 3 models for the choice of these parameters:

- choice of K_{min} based on the minimum number of gaussians in a gaussian mixture decomposition of the data;
- based on the square root of the number of patterns ($\{K_{min}, K_{max}\} = \{\frac{\sqrt{n}}{2}, \sqrt{n}\}$);
- or based on a linear transformation of the number of patterns ($\{K_{min}, K_{max}\} = \{\frac{n}{A}, \frac{n}{B}\}, A < B$).

The study compared how each model impacted the sparsity of the co-association matrix (and, thus, the space complexity) and the relative accuracy of the final clusterings. Both theoretical predictions and results revealed that the linear model produces the highest sparsity in the co-association matrix, under a dataset consisting of a mixture of Gaussians. Furthermore, it is true for both linear and square root models that the sparsity increases as the number of samples increases.

For real data sets, the performance of the three models became increasingly similar with the increase of the cardinality of the problem. It was found that the chosen granularity of the input partitions (K_{min}) is the variable with most impact, affecting both accuracy and sparsity. The authors reported this technique to have linear space and time complexity on benchmark data.

The number of samples of the data sets analysed in [29] was under 10^4. Furthermore, it should be noted that the remarks concerning the sparsity of the co-association matrix in the aforementioned study refer to the number of non-zero elements in the matrix and doesn't take into account extra data structures that accompany real sparse matrices implementations.

3.2 Clustering with large datasets

When large datasets, and big data, is in discussion, two perspectives should be taken into account [12]. The first deals with the applications where data is too large to be stored efficiently. This is the problem that streaming algorithms such as LOCALSEARCH [30] try to solve by analyzing data as it is produced, close to real-time processing. The other perspective is data that is actually stored for later processing. The latter is the perspective relevant to the present work, so it is further discussed below.

The flow of clustering algorithms typically involves some initialization step (e.g. choosing the number of centroids in K-Means) followed by an iterative process until some stopping criteria is met, where each iteration updates the clustering of the data [12]. In light of this, to speed up and/or scale up an algorithm, three approaches are available: (1) reduce the number of iterations, (2) reduce the number of patterns and/or features to process or (3) parallelizing and distributing the computation. The solutions for each of this approaches are, respectively, one-pass algorithms (e.g. CLARANS [31], BIRCH [32], CURE [33]), randomized techniques that reduce the input space complexity (e.g. PCA, CX/CUR [34]) and parallel algorithms (parallel K-Means [35], parallel spectral clustering [36]).

Parallelization can be attained by adapting algorithms to multi core CPU, GPU, distributed over several machines (a *cluster*) or a combination of the former, e.g. parallel and distributed processing using GPU in a cluster of hybrid workstations. Each approach has its advantages and disadvantages. The CPU approach has access to a larger memory but the number of computation units is reduced when compared with the GPU or cluster approach. Furthermore CPUs have advanced techniques such as branch prediction, multiple level caching and out of order execution - techniques for optimized sequential computation. GPU have hundreds or thousands of computing units but typically the available device memory is reduced which entails an increased overhead of memory transfer between host (workstation) and device (GPU) for computation of large datasets. In addition, it's harder to scale the above solutions for even bigger datasets. On the other hand, GPUs can be found on commodity computers, from laptops to workstations. A cluster offers a parallelized and distributed solution, which is easier to scale. According to [12], the two algorithmic approaches for cluster solutions are (1) memory-based, where the problem data fits in the main memory of the machines of the cluster and each machine loads part of the data; or (2) disk-based, comprising the widely used MapReduce framework capable of processing massive amounts of data in a distributed way. The main disadvantage is that there is a high communication and memory I/O cost to pay. Communication is usually done over the network with TCP/IP, which is several orders of magnitude slower that the direct access of the CPU or GPU to memory (host or device).

The present work is oriented towards GPU based parallelization, since GPUs are an easily accessible commodity and since the goals of the dissertation are oriented towards computation on a single machine. Taking that into consideration, this section starts by covering a review of the General Purpose computing on GPUs paradigm. It goes on to review a GPU parallel version of the K-Means algorithm and a GPU parallel approach for performing Single-Link clustering.

3.2.1 General Purpose computing on Graphical Processing Units

Using GPU for other applications other than graphic processing, commonly known as GPGPU, has become a trend in recent years. GPU present a solution for ”extreme-scale, cost-effective, and power-efficient high performance computing” [37]. Furthermore, GPU are typically found in consumer desktops and laptops, effectively bringing this computation power to the masses.

GPUs were typically useful for users that required high performance graphics computation. Other applications were soon explored as users from different fields realized the high parallel computation power of these devices. However, the architecture of the GPUs themselves has been strictly oriented toward the graphics computing until recently as specialized GPU models (e.g. NVIDIA Tesla) have been designed for data computation.

GPGPU application on several fields and algorithms has been reported with significant performance increase, e.g. application on the K-Means algorithm [38, 39, 40, 41], hierarchical clustering [42, 43], document clustering [44], image segmentation [45], integration in Hadoop clusters [46, 47], among other applications.

Current GPUs pack hundreds of cores and have a better energy/area ratio than traditional infrastructure. GPU work under the SIMD framework, i.e. all the cores in the device execute the same code at the same time and only the data changes over time.

Programming GPUs

In the very beginning of GPGPU, programming was done directly through graphics APIs. Programming for GPUs was traditionally done within the paradigm of graphics processing, such as DirectX and OpenGL. If researchers and programmers wanted to tap into the computing power of a GPU they had to learn and use these APIs and frameworks, which is a challenging task since their general problems had to be modelled to the graphics-oriented primitives [48]. With the appearance of DirectX 9, shader programming languages of higher level became available (e.g. C for graphics, DirectX High Level Shader Language, OpenGL Shading Language), but they were still inherently graphics programming languages, where computation must be expressed in graphics terms.

More recent programming models, such as CUDA and OpenCL, removed a lot of that burden by exposing the power of GPUs in a way closer to traditional programming. At the time of writing, the major programming models used for computation in GPU are OpenCL and CUDA. While the first is widely available in most devices the later is only available for NVIDIA devices.

As Google’s MapReduce computing model has increasingly become a standard for scalable and distributed computing over big data, attempts have been made to port the model to the GPU [49, 50, 51]. This translates in using the same programming model over a wide array of computing platforms.

OpenCL vs CUDA

At the moment of writing the most mature programming models are CUDA and OpenCL. CUDA was appeared first is the most mature of the two and is supported only by NVidia devices. OpenCL has the advantage of portability, but that comes with issues of performance portability. CUDA performs well since it was designed alongside with the hardware architecture itself. Both models are, in fact, very similar and literature suggests that porting the code from one to the other requires minimal changes [52, 53]. Literature also reports that performance from CUDA is better than OpenCL even for equivalent code.

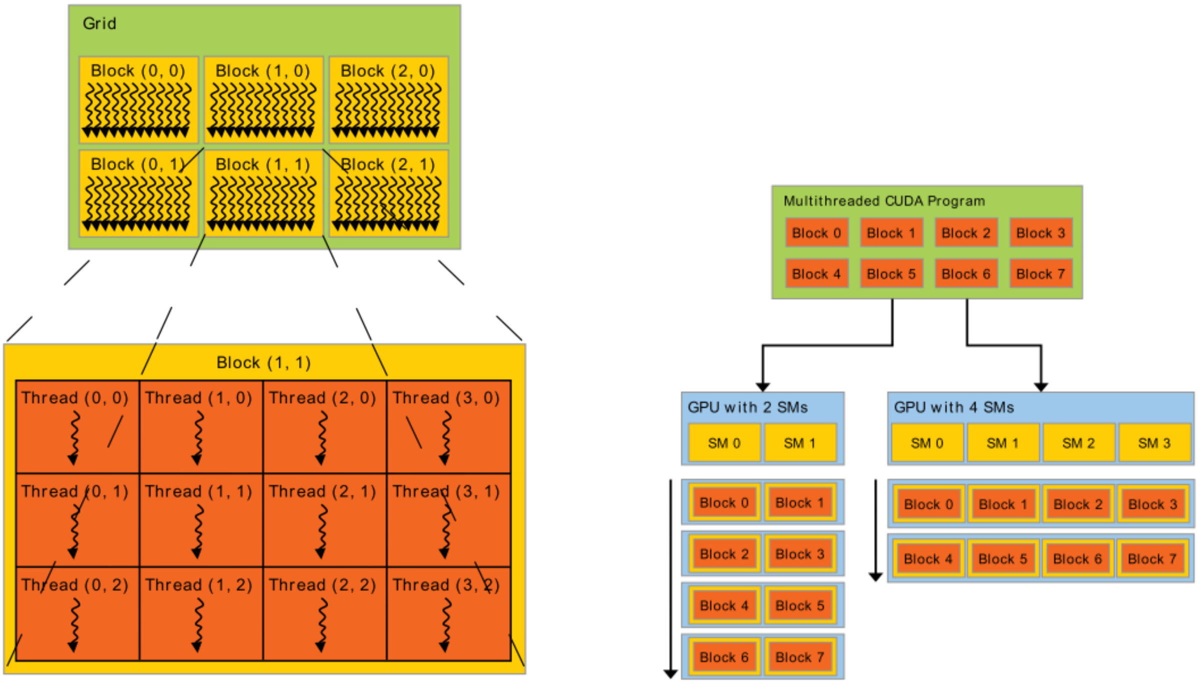
Overview of CUDA

This section presents an overview of the CUDA programming model and its main concepts and peculiarities. For a more thorough and extensive explanation of these topic, the CUDA C Programming Guides [54], the source of the present review, should be consulted. A GPU is constituted by one or several streaming processors (or multiprocessor). Each of this processors contains several simpler processors, each of which execute the same instruction at the same time at any given time. In the CUDA programming model, the basic unit of computation is a *thread*. Threads are grouped into *blocks* which are part of the block *grid*. The number of threads in a block is typically higher than the number of processors in a multiprocessor. For that reason, the hardware automatically partitions threads in a block into smaller batches, called *warps*. This hierarchy is represented in Figure 3.1a. The computation of one block is independent from other blocks and each block is scheduled to one multiprocessor, which means that more multiprocessors results in more blocks being processed at the same time, as represented in Figure 3.1b.

Block configuration can be multidimensional, up to and including 3 dimensions. Furthermore, there is a limit to the amount of threads in each dimension that varies with the version of CUDA being used, e.g. for GPUs with CUDA compute capability 2.x the number of threads is 1024 for the x- or y-dimensions, 64 for the z-dimension, an overall maximum number of threads is 1024 and a warp size of 32 threads. For the previous example, it's wise for the number of threads used in a block to be a multiple of 32 to maximize processor utilization, otherwise some blocks will have processors that will do no work.

Depending on the architecture, GPUs have several types of memories. Accessible to all processors (and threads) are the global memory, constant memory and texture memory, of which the last two are read-only. Blocks share a smaller but significantly faster memory called shared memory, which is a memory inside a multiprocessor to which all processors have access to, which enables inter-thread communication inside a block. Lastly, each thread as access to local memory. Local memory resides in global memory space and has the latency for read and write operations. However, if the thread is using only single variables or constant sized arrays since it stores them in the register space, which is very fast. If the memory used exceeds the available register space the local memory used. This memory hierarchy is represented in Figure 3.2a.

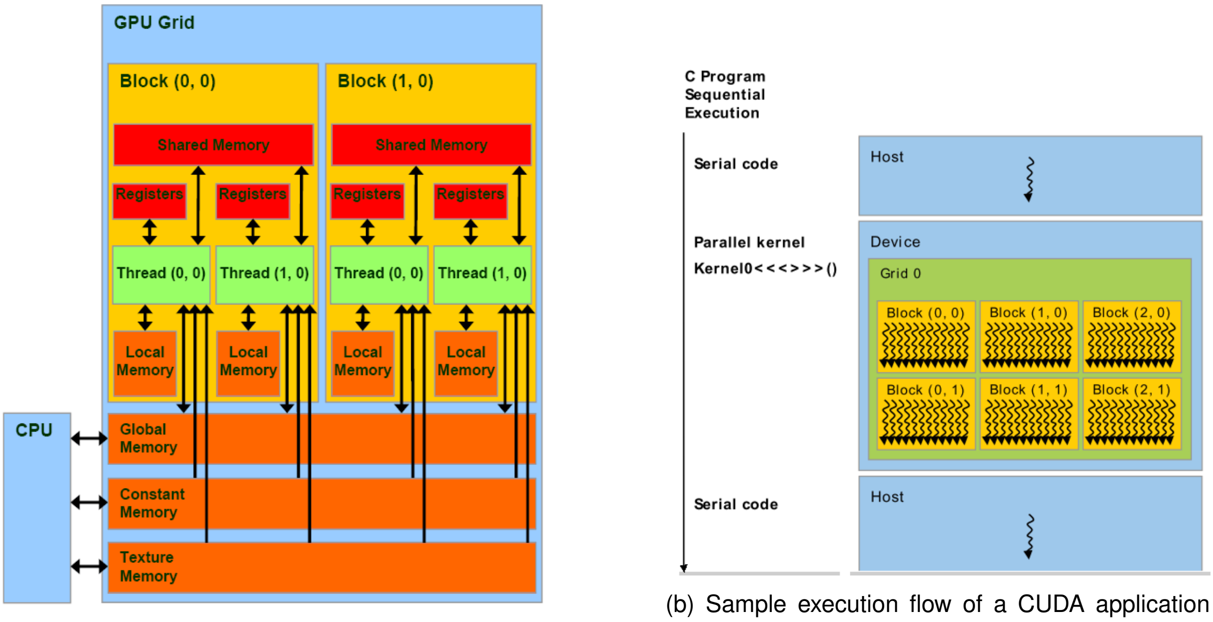
The typical flow of a CUDA application (and, typically, any modern GPU application) is explained in this paragraph and can be observed in Figure 3.2b. First, the host CPU transfers any necessary data



(a) Thread hierarchy [54].

(b) Automatic scalability [54].

Figure 3.1: 3.1a represents the thread hierarchy and 3.1b shows how the distribution of thread blocks is automatically scaled with the increase of the number of multiprocessors.



(a) Memory model used by CUDA [54].

(b) Sample execution flow of a CUDA application [54].

Figure 3.2

to the device memory (global, texture or constant) and is responsible for setting up the device code execution, which entails selecting the *kernel* (the function that will run on the GPU processors) and the thread topology (configuration of threads in a block and blocks in the grid). The next phase is simply the device executing the kernel. Finally, the host CPU will transfer back the results from the device. It should be noted that the latest architectures support *Dynamic Parallelism*. This functionality allows the device to start other kernels without the intervention of the host CPU, which would alter the typical execution flow explained above if used. It can be particularly useful if several kernels have to be executed in an application from the same input data but with dependencies. In such a scenario, a block of the second kernel could be executed as soon as all dependencies from the first kernel were met, effectively cutting overheads for kernel calling from the host CPU.

3.2.2 Parallel K-Means

K-Means is an obvious candidate to generate the ensemble of the first step of EAC. Other algorithms can be used, but due to its simplicity it is often used to produce ensembles varying the number of centroids to be used. Furthermore, it is a very good candidate for parallelization. As explained in chapter 2, K-Means has two main steps: labeling and update.

Several parallel implementations of this algorithm for the GPU exist [38, 39, 40, 45, 55] and all report significant speedups relative to their sequential counterparts in certain conditions, usually after the input data set goes above a certain cardinality, dimensionality or number of clusters threshold. The first step is inherently parallel as the computation of the label of the i-th pattern is not dependent on any other pattern, but only on the centroids. Two possible approaches to parallelize this step on the GPU are possible, a centroid-centric or a data-centric [38]. In the former each thread is responsible for a centroid and will compute the distance from its centroid to every pattern. These distances must be stored and, in the end, the patterns are assigned the closest centroid. In the later, each thread will compute the distance from one or more data points to every centroid and determines to which centroid they belong to (their labels). This strategy has the advantage of using less memory since it doesn't need to store all the pair-wise distances to perform the labeling - it only needs to store the best distance for each pattern. According to [38], the former approach is suitable for devices with a low number of processors so as to stream the data to each one while the later is better suited to devices with more processors.

The approach taken in [35] only parallelizes the labeling stage and takes a data-centric approach to the problem. Each thread computes the distance from a set of data points to every centroid and determines the labels. The remaining steps are performed by the host CPU. This study reported speed ups up to 14, for input datasets of 500000 points. Furthermore, it should be noted that the speed up was measured against a sequential version with all C++ compiler optimizations turned on, including vector operations (which, by themselves, are a way of parallelizing computation). The parallelized algorithm's flow can be observed in Figure 3.3.

It should be noted that the literature reports that the performance of K-Means using Dynamic Parallelism is slightly worse than that of its standard GPU counterpart [56].

3.2.3 Parallel Single-Link Clustering

Single-Link (SL) is an important step in the EAC chain. Given the new similarity metric (how many times a pair of patterns are clustered together in the ensemble), SL provides an intuitive way of obtaining the final partition: patterns that are clustered together often in the ensemble should remain clustered

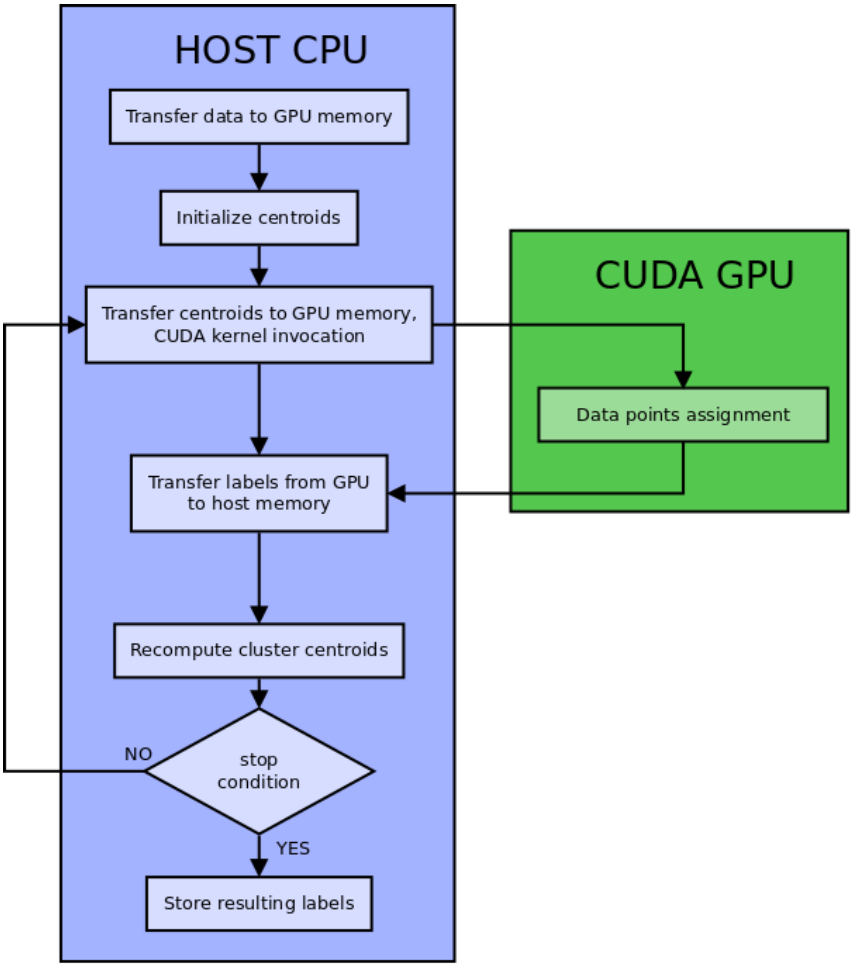

HOST CPU
Transfer data to GPU memory
Initialize centroids
Transfer centroids to GPU memory, CUDA kernel invocation
Transfer labels from GPU to host memory
Recompute cluster centroids
stop condition
NO
YES
Store resulting labels
CUDA GPU
Data points assignment

Figure 3.3: Flow execution of the GPU parallel K-Means algorithm.

together in the final solution.

SL is not easily parallelized since, typically, a new cluster generated at each step may include the one generated in the previous iteration. The most parallelizable part is the computation of the pair-wise similarity matrix, which is only useful if the input is raw data instead of a similarity matrix as in the case of EAC. The relationship between SL and the Minimum Spanning Tree problem explained in chapter 2 is the key to parallelize it. If one takes this approach for solving the SL problem, it becomes easier to parallelize it since parallel MST algorithms are abundant in literature [57, 58, 59]. The same approach for extracting the final clustering in EAC was used in [60].

Algorithm for finding Minimum Spanning Trees

There are several algorithms for computing an MST. The most famous are Kruskal [61], Prim [62] and Borůvka [63]. Borůvka's algorithm is also known as Sollin's algorithm. The first two are mostly sequential, while the later has the highest potential for parallelization, specially in the first iterations. As such, even though GPU parallel variants of Kruskal's [58] and Prim's [64] algorithms exist, the focus will be on Borůvka's.

Several parallel implementations of this algorithm for the GPU exist, e.g. [57], [65] and [59]. Sousa et al. [59] provides a more in-depth review over the current state of the art of MST solvers for the GPU and proposes an algorithm reported to be the fastest, as to the moment of writing. This section will review the algorithm proposed in [59], referred to as *Sousa2015* from henceforth.

CSR format

Sousa2015 takes in a graph as input, represented in the CSR format (a format used for sparse matrices). This representation is equivalent to having a square matrix G with zeroed diagonal where the g_{ij} element of the matrix is the weight of the link connecting the node i with the node j. This format is represented in Fig. 3.4. It requires three arrays to fully describe the graph:

- a *data* array containing all the non-zero values, where values from the same row appear sequentially from left to right and top to bottom, i.e. in *row-major* order, e.g. if the first row has 20 non-zero values, then the first 20 elements from this array belong to the first row;
- a *indices* array of the same size as *data* containing the column index of each non-zero value;
- a *indptr* array of the size of the number of rows containing a pointer to the first element in the *data* and *indices* arrays that belongs to each row, e.g. if the $i-th$ element (row) of *indptr* is k and it has 10 values, then all the elements from k to $k+10$ in *data* belong to the $i-th$ row.

Within the algorithm's context, the three arrays are denominated as *first_edge* (previously *indptr*, *destination* (*indices*) and *weight* (*data*). Although these three arrays can completely describe a graph, the algorithm uses an extra array *outdegree* that stores the number of non-zero values of each row and can be deduced from the *first_edge* array. The length and purpose of each of this arrays:

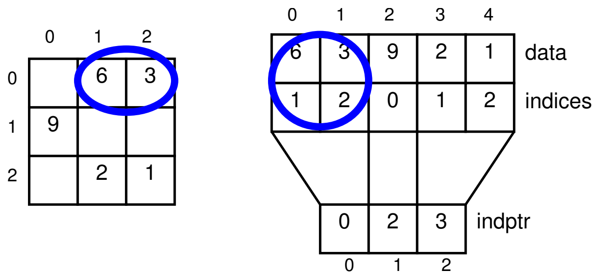


Figure 3.4: Correspondence between a sparse matrix and its CSR counterpart.

- *first_edge* is an array of size $\|V\|$, where the *i-th* element points to the first edge corresponding to the *i-th* edge.
- *outdegree* is an array of size $\|V\|$, where the *i-th* element contains the number of edges attached to the *i-th* edge.
- *destination* is an array of size $\|E\|$, where the *j-th* element points to the destination vertex of the *j-th* edge.
- *weight* is an array of size $\|E\|$, where the *j-th* element contains the weight of the *j-th* edge.

V is the number of vertices and E is the number of edges. The number of edges is duplicated to cover both directions, since the algorithm works with undirected graphs. This basically means that instead of using the the upper (of lower) triangular matrix (which can also completely describe the graph), it uses the complete one resulting in double redundancy of each edge. The edges in the *destination* array are grouped together by the vertex they originate from, e.g. if edge j is the first edge of vertex i and this vertex has 3 edges, then edges $\{j, j+1, j+2\}$ are the outgoing edges of vertex i.

Steps of the algorithm

Within the context of the algorithm the *id* of a vertex is its index in the *first_edge* array, the *color* is related to the *ids* and the *successor* of a vertex is the destination vertex of one of its edges. The algorithm's flow is represented in Figure 3.5 and its main steps are explained below:

1. *Find minimum edge per vertex*: select and store the minimum weighted edge for each vertex and resolve same weight conflict by picking the edge with lower destination vertex id.
2. *Remove mirrored edges*: a mirrored edge the successor of its destination vertex is its origin vertex. All mirrored edges are removed from the selected edges in the first step. All edges that are not removed are added to the resulting MST.
3. *Initialize and propagate colors*: this step is responsible for identifying connected components so the graph may be contracted. Each connected component will be a super-vertex in the contracted graph, which means it will be a single vertex that is representing a subgraph. Each vertex is initialized with the same color of its successor's id. If a vertex has no successor because that edge

was removed in the previous step than its color is initialized with its own id. The colors are then propagated by setting each vertex's color to the color of its successor, until convergence.

4. *Create new vertex ids*: only the super-vertices will be propagated to the next iteration but they'll have new ids for building the new contracted graph. The new ids will range from 0 to s, where s is the number of super-vertices. The vertex that will represent a super-vertex is the vertex whose color is its own id. The representative vertices will take the new ids in increasing order according to their own ids in increasing order, e.g. vertex 2 is the representative with lowest id so it will have the id 0 in the contracted graph. This step relied on the *exclusive scan* operation, which is explained in section 3.2.3.

5. *Count, assign, and insert new edges*: the final step is where the final operations for building the contracted graph are performed. The algorithm will count the number of edges that each super-vertex has connecting to other super-vertices, i.e. the connections between subgraphs. This is simply accomplished by selecting every edge whose origin and destination colors are distinct. For each such edge the corresponding positions of the origin and destination super-vertices in a new *outdegree* array are incremented with an atomic operation. The new *first_edge* array is obtained from performing an exclusive scan over *outdegree*. The next step is to assign and insert edges in the contracted graph. Once again, the algorithm will determine which edges will be in the contracted graph by checking the origin and destination colors. A copy *top_edge* of the new *first_edge* array is done to keep track of where to insert the new edges. When an edge is assigned to a super-vertex it is inserted (in the new *weight* and *destination* arrays) in the position specified in *top_edge* and the algorithm increments *top_edge* so the next edge will not overwrite the previous one. The increment is done with an atomic operation since multiple edges can be assigned to the same super-vertex and the insertion of all edges is being done in parallel. It should be noted that duplicated edges are note removed, i.e. several distinct connections between two super-vertices can be kept, since the author considers that the benefit of doing so does not outweigh the computational overhead.

It should be noted, however, that this algorithm doe not support unconnected graphs, i.e., it is not able to output a forest of MSTs. Upon contact, the author reported that a solution to that problem is, on the step of building the flag array, only mark a vertex if it is both the representative of its supervertex and has at least one neighbour.

Exclusive scan

The *scan* operation is one of the fundamental building block of parallel computing. Furthermore, two of the steps of the Borůvka variant of [59] are performed with an exclusive scan where the operation is a sum. To illustrate the functioning of the exclusive scan, let's consider the particular case where the operation of the scan is the sum and the identity (the value for which the operation produces the same output as the input) is naturally 0. Let's further consider the input array to be the sequence

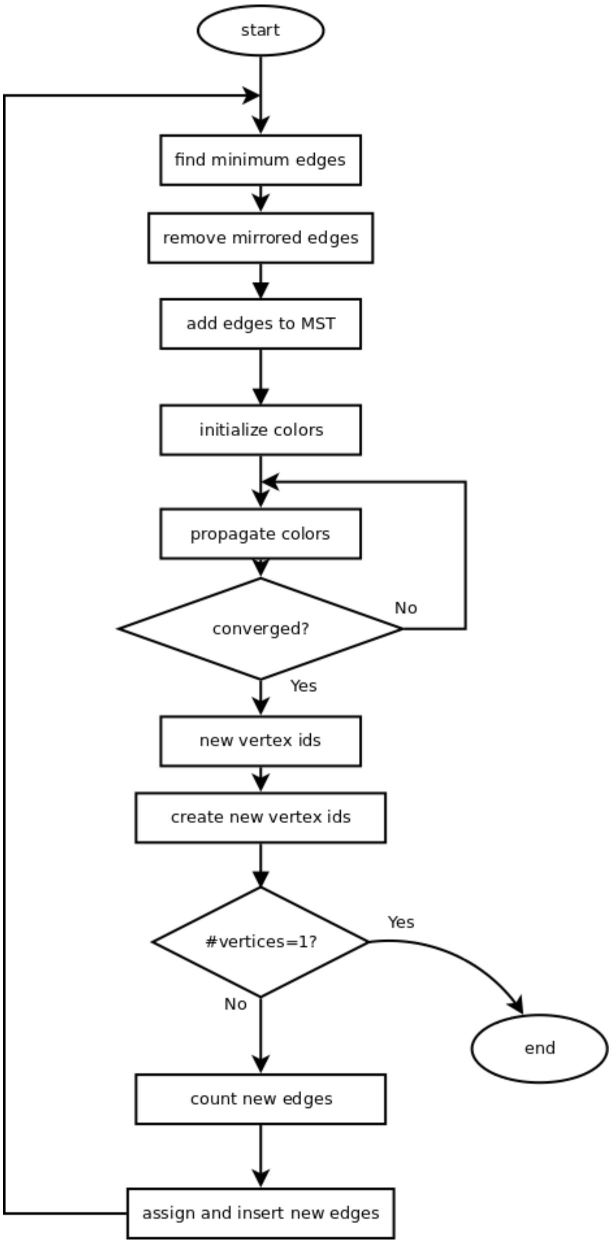


Figure 3.5: Flow execution of Sousa2015.

$[0, 1, 2, 3, 4, 5, 6, 7]$. Then the output will be $[0, 0, 1, 3, 5, 8, 13, 19]$. The first element of the output will be the identity (if it were an inclusive scan, it would be the first element itself). The second element is the sum between the first element of the input array and the first element of the output array, the third element is the sum between the second element of the input array with second element of the output array, and so on. This algorithm seems highly sequential in nature each element of the output array depends on the previous one, but two main parallel versions can be found in literature: Hillis and Steele's [66] and Blelloch's [67]. The two approaches focus on distinct efforts: the former focus on optimizing the number of steps [66] while the later focus on optimizing the amount of work done [67]. The focus will be on the Blelloch's algorithm.

Blelloch's algorithm is comprised by two main phases: the *reduce phase* (or *up-sweep*) and the *down-sweep phase*. The algorithm is based on the concept of *balanced binary trees* [1], but it should be noted that no such data structure is actually used. An in-depth explanation of this concept and how it relates to the algorithm being reviewed falls outside the scope of the present work and it is recommended that the reader consult [1] or [67] for such details.

During the reduce phase (see Figure 3.6), the algorithm traverses the tree and computes partial sums at the internal nodes of the tree. This operation is also known as a parallel reduction due to the fact that the total sum is stored in the root node (last node) of the tree [1].

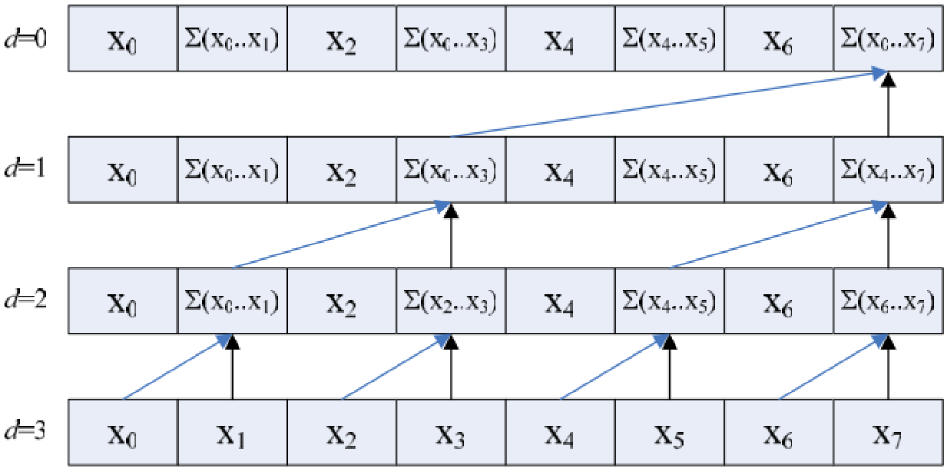
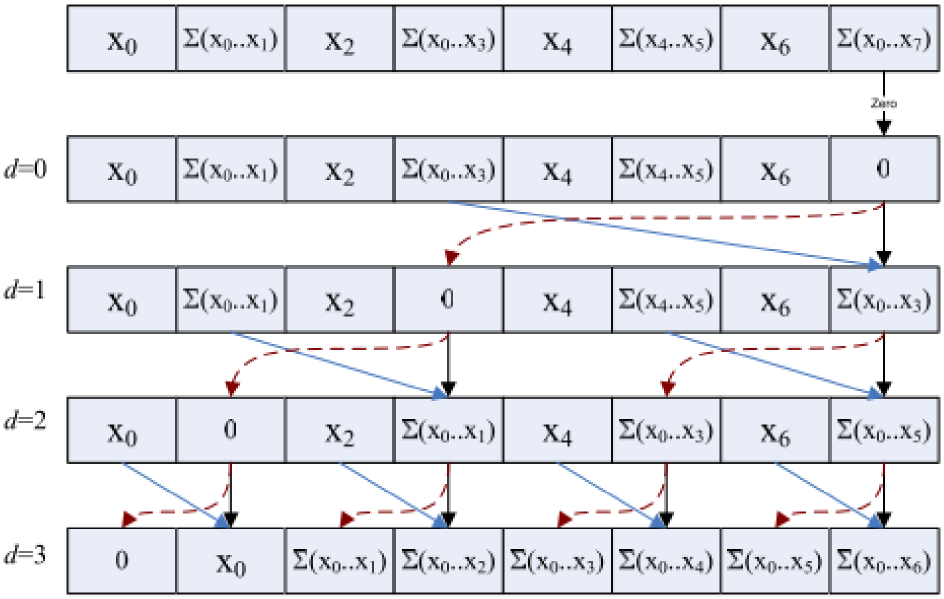


Figure 3.6: Representation of the reduce phase of Blelloch's algorithm [1]. *d* is the level of the tree and the input array can be observed at $d = 0$.

In the down-sweep phase (see Figure 3.7) the algorithm traverses back the tree. During the traversal, and using the partial sums calculated in the reduce phase, it builds the scan in place (overwriting the result of the reduce phase).

This the computational complexity of this algorithm is higher than that of its sequential counterpart. The sequential version has a $O(n)$ computational complexity since it only goes through the input array once and performs exactly n additions. Blelloch's algorithm, in the other hand, performs $2(n-1)$ additions in the reduce phase and $n-1$ swaps in the down-sweep phase. However, since it is a parallel algorithm and the computation will be distributed across several processing units, it still performs better. It should also be noted that, as described, the algorithm supports input arrays of a size that is a power of 2.



	x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_0..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_0..x_7)$
d=0	x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_0..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	0
d=1	x_0	$\Sigma(x_0..x_1)$	x_2	0	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_0..x_3)$
d=2	x_0	0	x_2	$\Sigma(x_0..x_1)$	x_4	$\Sigma(x_0..x_3)$	x_6	$\Sigma(x_0..x_5)$
d=3	0	x_0	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_0..x_4)$	$\Sigma(x_0..x_5)$	$\Sigma(x_0..x_6)$

Figure 3.7: Representation of the down-sweep phase of Blelloch's algorithm [1]. *d* is the level of the tree.

However, [1] explains how to overcome this limitation and also offers an implementation for CUDA and hardware specific optimizations.

3.3 Quantum clustering

The field of quantum clustering has shown promising results regarding potential speedups in several tasks over their classical counterparts. At the moment of writing, two major approaches for the concept of quantum clustering were found in the literature. The first is the quantization of clustering methods to work in quantum computers. This translates in converting algorithms to work partially or totally on a different computing paradigm, with support of quantum circuits or quantum computers. Both [68] and [69] show how the quantum paradigm can be used for speedups in the machine learning algorithms, with the possibility of obtaining exponential speed-ups. The quantization approach falls outside the scope of this dissertation, but [70] offers a thorough review on the state of the art of machine learning in the quantum paradigm. Many of the approaches for these quantizations make use of Groover's database search algorithm [71], or a variant of it, e.g. [72]. Most literature on this approach is also mostly theoretical since the physical requirements still don't exist for this methods to be tested. This approach can be seen as part of the bigger problem of quantum computing and quantum information processing. An alternative to using real quantum systems would be to simulate them. However, simulating quantum systems in classical computers is a very hard task by itself and literature suggest is not feasible [73].

Computational intelligence is the second approach, i.e. to use algorithms that muster inspiration from quantum analogies. A study of the literature reveals that this approach typically further divides itself into two concepts [74]. One comprehends the algorithms based on the concept of the qubit, the quantum analogue of a classical bit with interesting properties found in quantum objects. Several algorithms have been modeled after this concept, often also mustering inspiration from evolutionary genetic algorithms successfully applied in several areas:

- tackling the Knapsack problem as described in [75] and its improved version [76];
- solving the traveling salesman problem [77];
- two implementations of a Quantum K-Means algorithm [78, 79];
- a Quantum Artificial Bee Colony algorithm [80, 74];
- two approaches for Fuzzy C-Means (FCM), namely Quantum New Weighted Fuzzy C-Means (QNW-FCM) [74] and Quantum Fuzzy C-Means (QFCM) [81, 74];
- a novel quantum inspired clustering technique based on manifold distance [82];
- a Quantum-inspired immune clonal clustering algorithm based on watershed [83].

The other approach models data as a quantum system and uses Schrödinger's equation in some way, usually to evolve the data modeled particle system into a solution. This has been applied in an optimization problem for electromagnetics using a quantum particle swarm [84] and also on several clustering algorithms:

- the Quantum Clustering [85] algorithm treats patterns as quantum particles whose potential is computed with Schrödinger's equation and the system is evolved with the Gradient Descent method;
- Dynamic Quantum Clustering [86] is a variation of [85] that uses Schrödinger's equation to evolve the system;
- a Quantum Clustering [85] based Fuzzy C-Means approach [87];
- QPSO+FCM, a Quantum-behaved Particle Swarm Optimization (QPSO) [88] based approach to Fuzzy C-means [89].

For more information on quantum inspired algorithms, the reader is referred to [90] which offers a thorough survey on the state of the art of quantum inspired computation intelligence algorithms. The following two sections contain a brief overview of the concept of the qubit and the description of an algorithm that takes this approach. After that, an algorithm that follows the Schrödinger's equation approach is reviewed.

3.3.1 The quantum bit approach

The quantum bit

To understand the workings of the algorithms based on the concept of the qubit, it is useful to cast some insight about its properties and functioning. This section has the purpose to provide a brief introduction to this topic. An extended and in-depth review of this and related topics can be found in [91]. The qubit is a quantum object with certain quantum properties such as entanglement and superposition. Within the context of the studied algorithm, the only property used is superposition. A qubit can have any value between 0 and 1 (superposition property) until it is observed, which is when the system collapses to either state. However, the probability with which the system collapses to either state may be different. The superposition property or linear combination of states can be expressed [78] as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where ψ is an arbitrary state vector and α, β are the the probability amplitude coefficients of basis states $|0\rangle$ and $|1\rangle$, respectively. The Dirac bra ket notation is emplyed, where the *ket* $|.\rangle$ corresponds to a column vector. The basis states correspond to the spin of the modeled particle (in this case, a ferminion, e.g. electron). The coefficients are subjected to the following normalization:

$$|\alpha|^2 + |\beta|^2 = 1$$

where $|\alpha|^2$, $|\beta|^2$ are the probabilities of observing states $[0]$ and $[1]$, respectively. α and β are complex quantities and represent a qubit:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Moreover, a qubit string may be represented by:

$$\left[\begin{array}{c|c|c} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \end{array}\right]$$

The probability of observing the state $|000\rangle$ will be $|\alpha_1|^2 \times |\alpha_2|^2 \times |\alpha_3|^2$ To use this model for computing purposes, black-box objects called *oracles* are used. Oracles are important to understand quantum speedups. They can be understood as subroutines that cannot be usefully examined or unknown physical systems with properties one would like to estimate that perform a quantum operation on a qubit string [92]. Within the context of the present work an oracle is an abstraction for the programmer. It is an object which can be called and changes state (which can be observed) as a consequence. For the purpose of the following sections, the concept of the oracle is more related to *oracles with internal randomness* [92] or, more simply, a probabilistic Turing machine, as in the case of [80].

Quantum K-Means

The Quantum K-Means (QK-Means) algorithm, as described in [78], is based on the classical K-Means algorithm. It extends the basic K-Means with concepts from quantum mechanics (the qubit) and evolutionary genetic algorithms.

Within the context of this algorithm, oracles contain strings of qubits and generate their own input by observing the state of the qubits. After collapsing, the qubit value becomes corresponds to a classical bit, with a binary value.

Ideally, oracles would contain actual quantum systems or simulate them - this would correctly account for the desirable quantum properties. As it stands, oracles aren't quantum systems or even simulate them and can be more appropriately described as random number generators. Each string of qubits represents a number, so the number of qubits in each string will define its precision. The number of strings chosen for the oracles depends on the number of clusters and dimensionality of the problem (e.g. for 3 centroids of 2 dimensions, 6 strings will be used since 6 numbers are required). Each oracle will represent a possible solution.

The algorithm has the following steps:

1. Initialize population of oracles
2. Collapse oracles
3. K-Means
4. Compute cluster fitness
5. Store
6. Quantum Rotation Gate
7. Collapse oracles
8. Quantum cross-over and mutation
9. Repeat 3-7 until generation (iteration) limit is reached

Initialize population of oracles The oracles are created in this step and all qubit coefficients are initialized with $\frac{1}{\sqrt{2}}$, so that the system will observe either state of the qubit with equal probability. This value is chosen taken into account the necessary normalization of the coefficients, as described in the previous section.

Collapse oracles Collapsing the oracles implies making an observation of each qubit of each qubit string in each oracle. This is done by first choosing a coefficient to use (either can be used), e.g. α. Then, a random value r between 0 and 1 is generated. If $\alpha \geq r$ then the system collapses to $[0]$, otherwise to $[1]$.

K-Means In this step we convert the binary representation of the qubit strings to base 10 and use those values as initial centroids for K-Means. For each oracle, classical K-Means is then executed until it stabilizes or reaches the iteration limit. The solution centroids are returned to the oracles in binary representation.

Compute cluster fitness Cluster fitness is computed using the Davies-Bouldin index for each oracle. The score of each oracle is stored in the oracle itself.

Store The best scoring oracle is stored.

Quantum Rotation Gate So far, the algorithm consisted of the classical K-Means with a complex random number generation for the centroids and complicated data structures, namely the oracles. This is the step that fundamentally differs from the classical version. In this step a quantum gate (in this case a rotation gate) is applied to all oracles except the best one. The basic idea is to shift the qubit coefficients of the least scoring oracles in the direction of the best scoring one so they'll have a higher probability of collapsing into initial centroid values closer to the best solution so far. This way, in future generations, we'll not initiate with the best centroids so far (which will not converge further into a better solution) but we'll be closer while still ensuring diversity (which is also a desired property of the genetic computing paradigm). In other words, we look for better solutions than the one we got before in each oracle while moving in the direction of the best we found so far.

The genetic operations of cross-over and mutation are both part of the genetic algorithms toolbox. [72] suggests that this operations may not be required to produce variability in the population of qubit strings. This is because, according to [76], use of the angle-distance rotation method in the quantum rotation operation produces enough variability, with a careful choice of the rotation angle. However, when used, their goal is to produce further variability into the population of qubit strings.

3.3.2 Schrödinger's equation approach

The other approach to clustering that gathers inspiration from quantum mechanical concepts is to use the Schrödinger equation. The algorithm under study was created by Horn and Gottlieb [93] and was later extended by Weinstein and Horn [94].

The first step in this methodology is to compute a probability density function of the input data. This is done with a Parzen-window estimator in [95, 94]. The Parzen-window density estimation of the input data is done by associating a Gaussian with each point, such that

$$\psi(\mathbf{x}) = \sum_{i=1}^{N} e^{-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}}$$

where N is the total number of points in the dataset, σ is the variance and ψ is the probability density estimation. ψ is chosen to be the wave function in Schrödinger's equation. The details of why this is fall outside of the scope of the present work and are explained in [94, 95, 85].

Having this information we'll compute the potential function $V(x)$ that corresponds to the state of minimum energy (ground state = eigenstate with minimum eigenvalue) [95], by solving the Schrödinger's equation in order of $V(x)$:

$$V(\mathbf{x}) = E + \frac{\frac{\sigma^2}{2}\nabla^2\psi}{\psi} = E - \frac{d}{2} + \frac{1}{2\sigma^2\psi}\sum_{i=1}^{N} \|\mathbf{x}-\mathbf{x}_i\|^2 \, e^{-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}}$$

And since the energy should be chosen such that ψ is the groundstate (i.e. eigenstate corresponding to minimum eigenvalue) of the Hamiltonian operator associated with Schrödinger's equation (not represented above), the following is true

$$E = -min\frac{\frac{\sigma^2}{2}\nabla^2\psi}{\psi}$$

With all of the above, $V(x)$ can be computed. This potential function is akin to the inverse of a probability density function. Minima of the potential correspond to intervals in space where points are together. So minima will naturally correspond to cluster centers [95]. However, it's very computationally intensive to compute $V(x)$ to the whole space, so the compututation of of the potential function is only done at the data points. This should not be problematic since clusters' centers are generally close to the data points themselves. Even so, the minima may not lie on the data points themselves. One method to address this problem is to compute the potential on the input data and converge this points toward some minima of the potential function. This is done with the gradient descent method in [95].

Another method [94] is to think of the input data as particles and use the Hamiltonian operator to evolve the quantum system in the time-dependant Schrödinger equation. Given enough time steps, the particles will converge to and oscillate around potential minima. This method makes the Dynamic Quantum Clustering algorithm. The nature of the computations involved in this algorithm make it a good candidate for parallelization techniques. [96] parallelized this algorithm to the GPU obtaining speedups of up to two magnitudes relative to an optimized multicore CPU implementation.

Bibliography

[1] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA Mark. *Gpu gems 3*, (April):1–24, 2007. URL `http://dl.acm.org/citation.cfm?id=1407436`.

[2] Anil K Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, 2010. ISSN 01678655. doi: 10.1016/j.patrec.2009.09.011. URL `http://dx.doi.org/10.1016/j.patrec.2009.09.011`.

[3] a. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3): 264–323, 1999. ISSN 03600300. doi: 10.1145/331499.331504.

[4] Edgar Anderson. The species problem in Iris. *Annals of the Missouri Botanical Garden*, pages 457–509, 1936.

[5] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7 (2):179–188, 1936.

[6] Daniel Boley. Principal Direction Divisive Partitioning. *Data Mining and Knowledge Discovery*, 2(4):325–344, 1998. ISSN 1384-5810. doi: 10.1023/A:1009740529316. URL `http://dx.doi.org/10.1023/A:1009740529316$\delimiter"026E30F$nhttp://www.springerlink.com/content/w15313n737603612/`.

[7] Michael Steinbach, George Karypis, Vipin Kumar, and Others. A comparison of document clustering techniques. *KDD workshop on text mining*, 400(1):525–526, 2000. doi: 10.1.1.125.9225.

[8] Marie Chavent. A monothetic clustering method. *Pattern Recognition Letters*, 19(11):989–996, 1998. ISSN 01678655. doi: 10.1016/S0167-8655(98)00087-7.

[9] James C. Bezdek, Robert Ehrlich, and William Full. FCM: The fuzzy ¡i¿c¡/i¿-means clustering algorithm. *Computers & Geosciences*, 10(2–3):191–203, 1984. ISSN 0098-3004. doi: 10.1016/0098-3004(84)90020-7. URL `http://www.sciencedirect.com/science/article/pii/0098300484900207`.

[10] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. PEGASUS : Mining Peta-Scale Graphs. *Knowledge and Information Systems*, 27(2):303–325, 2011.

[11] J B MacQueen. Some Methods for classification and Analysis of Multivariate Observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press*, volume 1, pages 281–297, 1967.

[12] Charu C Aggarwal and Chandan K Reddy. *Data clustering: algorithms and applications*. CRC Press, 2013. ISBN 9781466558229.

[13] D. Arthur, D. Arthur, S. Vassilvitskii, and S. Vassilvitskii. k-means++: The advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 8: 1027–1035, 2007. doi: 10.1145/1283383.1283494. URL `http://portal.acm.org/citation.cfm?id=1283494`.

[14] Malay K Pakhira. A Modified k -means Algorithm to Avoid Empty Clusters. *International Journal of Recent Trends in Enineering*, 1(1), 2009.

[15] Ana Fred. Finding consistent clusters in data partitions. *Multiple classifier systems*, pages 309–318, 2001. URL `http://link.springer.com/chapter/10.1007/3-540-48219-9_31`.

[16] Peter H A Sneath and Robert R Sokal. Numerical taxonomy. *Nature*, 193(4818):855–860, 1962.

[17] R. Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method, 1973. ISSN 1460-2067. URL `http://comjnl.oxfordjournals.org/content/16/1/30.short`.

[18] J. C. Gower and G. J. S. Ross. Minimum Spanning Trees and Single Linkage Cluster Analysis. *Journal of the Royal Statistical Society*, 18(1):54–64, 1969.

[19] Daniel Müllner. Modern hierarchical, agglomerative clustering algorithms. (1973):29, 2011. URL `http://arxiv.org/abs/1109.2378`.

[20] J-L Starck and Fionn Murtagh. *Astronomical image and data analysis*. Springer Science & Business Media, 2007.

[21] Alexander Topchy, Anil K Jain, and William Punch. A mixture model for clustering ensembles. In *Society for Industrial and Applied Mathematics. Proceedings of the SIAM International Conference on Data Mining*, page 379. Society for Industrial and Applied Mathematics, 2004.

[22] Ana N L Fred and Anil K Jain. Combining multiple clusterings using evidence accumulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6):835–850, 2005.

[23] Hongjun Wang, Hanhuai Shan, and Arindam Banerjee. Bayesian cluster ensembles. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 4(1):54–70, 2011.

[24] Alexander Strehl and Joydeep Ghosh. Cluster Ensembles – A Knowledge Reuse Framework for. *Journal of Machine Learning Research*, 3:583–617, 2002. ISSN 1532-4435. doi: 10.1162/153244303321897735.

[25] Sandrine Dudoit and Jane Fridlyand. Bagging to improve the accuracy of a clustering procedure. *Bioinformatics*, 19(9):1090–1099, 2003. ISSN 13674803. doi: 10.1093/bioinformatics/btg038.

[26] You Wen Qian, William Cukierski, Mona Osman, and Lauri Goodell. Combined multiple clusterings on flow cytometry data to automatically identify chronic lymphocytic leukemia. *ICBBT 2010 - 2010 International Conference on Bioinformatics and Biomedical Technology*, pages 305–309, 2010. doi: 10.1109/ICBBT.2010.5478955.

[27] André Lourenço, Carlos Carreiras, Samuel Rota Bulò, and Ana Fred. ECG ANALYSIS USING CONSENSUS CLUSTERING. pages 511–515, 2009. URL `Lourenco2009`.

[28] André Lourenço and Ana Fred. Ensemble methods in the clustering of string patterns. *Proceedings - Seventh IEEE Workshop on Applications of Computer Vision, WACV 2005*, pages 143–148, 2007. doi: 10.1109/ACVMOT.2005.46.

[29] André Lourenço, Ana L N Fred, and Anil K. Jain. On the scalability of evidence accumulation clustering. *Proceedings - International Conference on Pattern Recognition*, 0:782–785, 2010. ISSN 10514651. doi: 10.1109/ICPR.2010.197.

[30] L O'Callaghan, N Mishra, A Meyerson, S Guha, and R Motwani. Streaming-data algorithms for high-quality clustering. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 685–694, 2002. doi: 10.1109/ICDE.2002.994785.

[31] Raymond T Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *Knowledge and Data Engineering, IEEE Transactions on*, 14(5):1003–1016, 2002.

[32] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD Record*, volume 25, pages 103–114. ACM, 1996.

[33] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. In *ACM SIGMOD Record*, volume 27, pages 73–84. ACM, 1998.

[34] Petros Drineas, Ravi Kannan, and Michael W Mahoney. Fast Monte Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition. *SIAM Journal on Computing*, 36 (1):184–206, 2006.

[35] Mario Zechner and Michael Granitzer. K-Means on the Graphics Processor: Design And Experimental Analysis. *International Journal on Advances in System and Measurements*, 2(2):224–235, 2009. doi: issn:1942-261x. URL `http://www.iariajournals.org/systems_and_measurements/sysmea_v2_n23_2009_paged.pdf`.

[36] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y Chang. Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(3):568–586, 2011.

[37] Linchuan Chen and Gagan Agrawal. Optimizing MapReduce for GPUs with Effective Shared Memory Usage. *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*, pages 199–210, 2012. doi:

10.1145/2287076.2287109. URL http://dl.acm.org/citation.cfm?doid=2287076.2287109$\delimiter"026E30F$nhttp://dl.acm.org/citation.cfm?id=2287109.

[38] Hong Tao Bai, Li Li He, Dan Tong Ouyang, Zhan Shan Li, and He Li. K-means on commodity GPUs with CUDA. *2009 WRI World Congress on Computer Science and Information Engineering, CSIE 2009*, 3:651–655, 2009. doi: 10.1109/CSIE.2009.491.

[39] Jiadong Wu and Bo Hong. An efficient k-means algorithm on CUDA. *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1740–1749, 2011. ISSN 1530-2075. doi: 10.1109/IPDPS.2011.331.

[40] Mario Zechner and Michael Granitzer. Accelerating k-means on the graphics processor via CUDA. *Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009*, pages 7–15, 2009. doi: 10.1109/INTENSIVE.2009.19.

[41] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering billions of data points using GPUs. *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–5, 2009. doi: 10.1145/1531666.1531668. URL http://portal.acm.org/citation.cfm?id=1531666.1531668.

[42] S. a Arul Shalom, Manoranjan Dash, Minh Tue, and Nithin Wilson. Hierarchical agglomerative clustering using graphics processor with compute unified device architecture. *2009 International Conference on Signal Processing Systems, ICSPS 2009*, pages 556–561, 2009. doi: 10.1109/ICSPS.2009.167.

[43] S. a. Arul Shalom and Manoranjan Dash. Efficient hierarchical agglomerative clustering algorithms on GPU using data partitioning. *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, pages 134–139, 2011. doi: 10.1109/PDCAT.2011.38.

[44] Zhanchun Gao, Enxing Li, and Yanjun Jiang. A gpu-based harmony k-means algorithm for document clustering. pages 2–5.

[45] J Sirotkovi, H Dujmi, and V Papi. K-Means Image Segmentation on Massively Parallel GPU Architecture. pages 489–494, 2012.

[46] Ranajoy Malakar and Naga Vydyanathan. A CUDA-enabled hadoop cluster for fast distributed image processing. *2013 National Conference on Parallel Computing Technologies, PARCOMPTECH 2013*, 2013. doi: 10.1109/ParCompTech.2013.6621392.

[47] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. HadoopCL: MapReduce on distributed heterogeneous platforms through seamless integration of hadoop and OpenCL. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, pages 1918–1927, 2013. doi: 10.1109/IPDPSW.2013.246.

[48] Marko J Miši, M Ĉ, and Milo V Tomaševi. Evolution and Trends in GPU Computing. *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 289–294, 2012.

[49] Feng Ji and Xiaosong Ma. Using Shared Memory to Accelerate MapReduce on Graphics Processing Units. *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 805–816, 2011. ISSN 1530-2075. doi: 10.1109/IPDPS.2011.80.

[50] Miao Xin and Hao Li. An implementation of GPU accelerated MapReduce: Using Hadoop with OpenCL for data- and compute-intensive jobs. *Proceedings - 2012 International Joint Conference on Service Sciences, Service Innovation in Emerging Economy: Cross-Disciplinary and Cross-Cultural Perspective, IJCSS 2012*, pages 6–11, 2012. doi: 10.1109/IJCSS.2012.22.

[51] Bingsheng He, Weibin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. *International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008. ISSN 03009440. doi: 10.1145/1454115.1454152. URL http://dl.acm.org/citation.cfm?id=1454152.

[52] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL. (1):12, 2010. doi: 10.1109/ICPP.2011.45. URL http://arxiv.org/abs/1005.2581.

[53] Ching Lung Su, Po Yu Chen, Chun Chieh Lan, Long Sheng Huang, and Kuo Hsuan Wu. Overview and comparison of OpenCL and CUDA technology for GPGPU. *IEEE Asia-Pacific Conference on Circuits and Systems, Proceedings, APCCAS*, pages 448–451, 2012. doi: 10.1109/APCCAS.2012.6419068.

[54] Nvidia. CUDA C Programming Guide. *Programming Guides*, (August), 2015.

[55] Reza Farivar, Daniel Rebolledo, and Ellick Chan. A parallel implementation of k-means clustering on GPUs. *on Parallel and*, pages 1–6, 2008. URL http://nguyendangbinh.org/Proceedings/IPCV08/Papers/PDP3663.pdf.

[56] Jeffrey DiMarco and Michela Taufer. Performance impact of dynamic parallelism on different clustering algorithms. *Spie*, page 87520E, 2013. ISSN 0277786X. doi: 10.1117/12.2018069. URL http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.2018069.

[57] Vibhav Vineet. Fast Minimum Spanning Tree for Large Graphs on the GPU by Fast Minimum Spanning Tree for Large Graphs on the GPU. 2009(August), 2009.

[58] Scott Rostrup, Shweta Srivastava, and Kishore Singhal. Fast and memory-efficient minimum spanning tree on the GPU. *International Journal of Computational Science and Engineering*, 8(1):21–33, 2013.

[59] Cristiano da Silva Sousa, Artur Mariano, and Alberto Proença. A Generic and Highly Efficient Parallel Variant of Boruvka 's Algorithm. URL https://github.com/Beatgodes/BoruvkaUMinho.

[60] Ana N L Fred and Anil K Jain. Data clustering using evidence accumulation. *Object recognition supported by user interaction for service robots*, 4, 2002. ISSN 1051-4651. doi: 10.1109/ICPR.2002.1047450.

[61] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[62] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[63] Otakar Borůvka. O jistém problému minimálním. 1926.

[64] Wei Wang, Yongzhong Huang, and Shaozhong Guo. Design and Implementation of GPU-Based Prim ' s Algorithm. *International Journal of Modern Education and Computer Science*, 4(July): 55–62, 2011.

[65] Pawan Harish, Vibhav Vineet, and P J Narayanan. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.

[66] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12): 1170–1183, 1986. ISSN 00010782. doi: 10.1145/7902.7903.

[67] Guy E Blelloch. Prefix Sums and Their Applications. *Computer*, pages 35–60, 1990. doi: 10.1.1. 47.6430. URL `http://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf`.

[68] Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. Quantum speed-up for unsupervised learning. *Machine Learning*, 90(February 2012):261–287, 2013. ISSN 08856125. doi: 10.1007/ s10994-012-5316-5.

[69] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411*, 2013. URL `http://arxiv.org/pdf/1307.0411.pdf$\delimiter"026E30F$npapers2://publication/uuid/2BDCBB85-812C-46E4-B506-73B9A41EBBC1`.

[70] Peter Wittek. *Quantum Machine Learning: What Quantum Computing Means to Data Mining*. Academic Press, 2014.

[71] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

[72] Nathan Wiebe, Ashish Kapoor, and Krysta Svore. Quantum Algorithms for Nearest-Neighbor Methods for Supervised and Unsupervised Learning. page 31, 2014. URL `http://arxiv.org/abs/1401.2142`.

[73] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1982. ISSN 00207748. doi: 10.1007/BF02650179.

[74] Ellis Casper and Chih-cheng Hung. Quantum Modeled Clustering Algorithms for Image Segmentation. 2(March):1–21, 2013. doi: 10.4156/pica.vol2.issue1.1.

[75] Kuk-Hyun Han and Jong-Hwan Kim. Genetic quantum algorithm and its application to combinatorial optimization problem. *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, 2, 2000. doi: 10.1109/CEC.2000.870809.

[76] Wenjie Liu, Hanwu Chen, Qiaoqiao Yan, Zhihao Liu, Juan Xu, and Yu Zheng. A novel quantum-inspired evolutionary algorithm based on variable angle-distance rotation. *2010 IEEE World Congress on Computational Intelligence, WCCI 2010 - 2010 IEEE Congress on Evolutionary Computation, CEC 2010*, 2010. doi: 10.1109/CEC.2010.5586281.

[77] H Talbi, A Draa, and M Batouche. A new quantum-inspired genetic algorithm for solving the travelling salesman problem. In *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on*, volume 3, pages 1192–1197 Vol. 3, 2004. doi: 10.1109/ICIT.2004.1490730.

[78] Ellis Casper, Chih-Cheng Hung, Edward Jung, and Ming Yang. A Quantum-Modeled K-Means Clustering Algorithm for Multi-band Image Segmentation. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, volume 1, pages 158–163, 2012. URL http://delivery.acm.org/10.1145/2410000/2401639/p158-casper.pdf?ip=193.136.132.10&id=2401639&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.F7A57B2C5B227641.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=476955365&CFTOKEN=55494231&__acm__=1423057410_0d77d9b5028cb3.

[79] Jing Xiao, YuPing Yan, Jun Zhang, and Yong Tang. A quantum-inspired genetic algorithm for k-means clustering. *Expert Systems with Applications*, 37:4966–4973, 2010. ISSN 09574174. doi: 10.1016/j.eswa.2009.12.017. URL http://ac.els-cdn.com/S095741740901063X/1-s2.0-S095741740901063X-main.pdf?_tid=f303a76c-ac71-11e4-be73-00000aacb35e&acdnat=1423056793_66291f279193fa69b86c93aecea405b0.

[80] Chih-Cheng Hung, Ellis Casper, Bor-chen Kuo, Wenping Liu, Edward Jung, Ming Yang, Xiaoyi Yu, Edward Jung, and Ming Yang. A Quantum-Modeled Artificial Bee Colony clustering algorithm for remotely sensed multi-band image segmentation. In *Geoscience and Remote Sensing Symposium (IGARSS), 2013 IEEE International*, pages 2501–2504. IEEE, 2013. ISBN 9781479911141.

[81] Chih-Cheng Hung, Ellis Casper, Bor-Chen Kuo, Wenping Liu, Xiaoyi Yu, Edward Jung, and Ming Yang. A Quantum-Modeled Fuzzy C-Means clustering algorithm for remotely sensed multi-band image segmentation. In *Geoscience and Remote Sensing Symposium (IGARSS), 2013 IEEE International*, pages 2501–2504. IEEE, 2013.

[82] Shenshen Liang, Cheng Wang, Ying Liu, and Liheng Jian. CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU. *Proceedings - 2009 IEEE Youth Conference on Information, Computing and Telecommunication, YC-ICT2009*, pages 415–418, 2009. doi: 10.1109/YCICT.2009.5382329.

[83] Yangyang Li, Nana Wu, Jingjing Ma, and Licheng Jiao. Quantum-inspired immune clonal clustering algorithm based on watershed. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE,

July 2010. ISBN 978-1-4244-6909-3. doi: 10.1109/CEC.2010.5586362. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5586362.

[84] Said M Mikki, Ahmed Kishk, and Others. Quantum particle swarm optimization for electromagnetics. *Antennas and Propagation, IEEE Transactions on*, 54(10):2764–2775, 2006.

[85] David Horn and Assaf Gottlieb. Algorithm for Data Clustering in Pattern Recognition Problems Based on Quantum Mechanics. *Physical Review Letters*, 88(1):1–4, 2001. ISSN 0031-9007. doi: 10.1103/PhysRevLett.88.018702. URL http://journals.aps.org/prl/abstract/10.1103/PhysRevLett.88.018702.

[86] Marvin Weinstein and David Horn. Dynamic quantum clustering: A method for visual exploration of structures in data. *Physical Review E*, 80(6):066117, December 2009. ISSN 1539-3755. doi: 10.1103/PhysRevE.80.066117. URL http://link.aps.org/doi/10.1103/PhysRevE.80.066117.

[87] Zhi-Hua Li, Shi-Tong Wang, and Jiangsu Wuxi. Quantum Theory: The unified framework for FCM and QC algorithm. In *Wavelet Analysis and Pattern Recognition, 2007. ICWAPR'07. International Conference on*, volume 3, pages 1045–1048. IEEE, 2007. ISBN 1424410665.

[88] Jun Sun, Bin Feng, and Wenbo Xu. Particle swarm optimization with particles having quantum behavior. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 325–331 Vol.1, 2004. doi: 10.1109/CEC.2004.1330875.

[89] Hao Wang Hao Wang, Shiqin Yang Shiqin Yang, Wenbo Xu Wenbo Xu, and Jun Sun Jun Sun. Scalability of Hybrid Fuzzy C-Means Algorithm Based on Quantum-Behaved PSO. *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, 2(Fskd), 2007. doi: 10.1109/FSKD.2007.507.

[90] a. Manju and M. J. Nigam. Applications of quantum inspired computational intelligence: A survey. *Artificial Intelligence Review*, 42:79–156, 2014. ISSN 02692821. doi: 10.1007/s10462-012-9330-6.

[91] Marco Lanzagorta and Jeffrey Uhlmann. *Quantum Computer Science*, volume 1. 2008. ISBN 9780511813870. doi: 10.2200/S00159ED1V01Y200810QMC002.

[92] David Rosenbaum and Aram W. Harrow. Uselessness for an Oracle Model with Internal Randomness. *arXiv:1111.1462*, pages 1–23, 2011. ISSN 15337146.

[93] David Horn, Tel Aviv, Assaf Gottlieb, Hod HaSharon, Inon Axel, and Ramat Gan. Method and Apparatus for Quantum Clustring, 2010.

[94] Marvin Weinstein and David Horn. Dynamic quantum clustering: a method for visual exploration of structures in data. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 80(6):1–15, December 2009. ISSN 1539-3755. doi: 10.1103/PhysRevE.80.066117. URL http://link.aps.org/doi/10.1103/PhysRevE.80.066117.

[95] David Horn and Assaf Gottlieb. The Method of Quantum Clustering. *NIPS*, (1), 2001. URL http://www-2.cs.cmu.edu/Groups/NIPS/NIPS2001/papers/psgz/AA08.ps.gz.

[96] Peter Wittek. High-performance dynamic quantum clustering on graphics processors. *Journal of Computational Physics*, 233:262–271, 2013. ISSN 00219991. doi: 10.1016/j.jcp.2012.08.048. URL `http://dx.doi.org/10.1016/j.jcp.2012.08.048`.