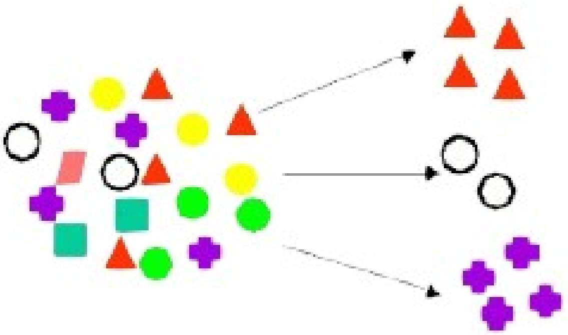


Efficient Evidence Accumulation Clustering for large datasets/big data

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 application 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 fail using sensor data, or even 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 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 EAC (Evidence Accumulation Clustering), belonging to the wider class of ensemble algorithms. 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 memory and computational complexities with parallelized computation and distributed memory. 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 in large datasets, while also accessing algorithmic solutions and parallelization techniques. The goal is to understand EAC's suitability for large datasets and find ways to respond to the challenges that that entails, in terms of speed and memory. The main objectives for this work are:

- Study integration of quantum inspired methods in EAC.
- Study integration of GPGPU paradigm in EAC.
- Devise strategies to reduce computation and memory complexities of EAC.
- Application of Evidence Accumulation Clustering in Big Data.
- Validation of Big Data EAC on real data

1.3 Contribution

The main contributions are the adaptation of the three distinct stages of the EAC toolchain to larger datasets. In particular, an efficient parallel version for 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, sparse matrices and a k-Nearest Neighbors scheme. 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 disk solution was implemented for dealing with large datasets in the final stage.

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, 4, 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 5 presents an interpretation and critical discussion of those results. Finally, chapter 6 concludes the dissertation. It also offers recommendations for future work.

Chapter 2

Clustering

2.1 The problem of clustering

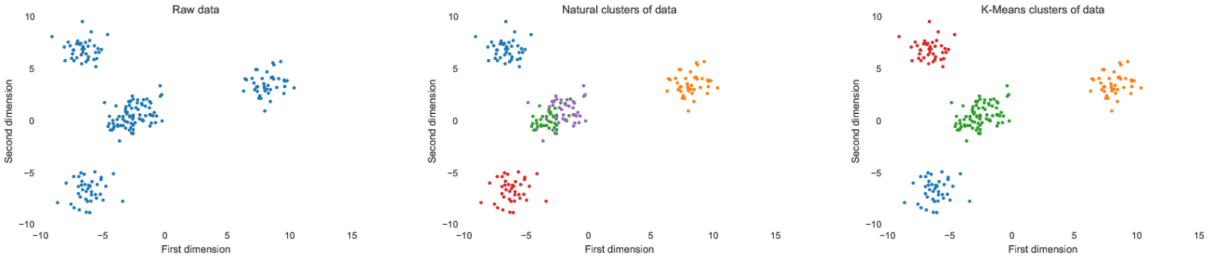
Advances in technology allow for the collection and storage unprecedented amount and variety of data. Since data is mostly stored electronically, it presents a potential for automatic analysis and thus creation of information and knowledge. A growing body of statistical methods aiming to model, structure and/or classify data already exist, e.g. linear regression, principal component analysis, cluster analysis, support vector machines, neural networks. 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. Unsupervised learning deals with unlabeled data and tries to solve the problem of clustering.

Cluster analysis is 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. And the methodology used is to group patterns that are similar by some metric (e.g. euclidean distance, Pearson correlation) and separate those that are dissimilar.

As an example, Figure 2.1a shows the plot of a simple synthetic dataset - a Gaussian mixture of 5 distributions. No extra information other than the position of the points is given, since clustering algorithms are unsupervised methods. Figure 2.1b presents the desired (or ”natural”) clustering for this given dataset. Figure 2.1c presents the clusters given by the K-Means algorithm with an initialization of 4 clusters. The number of clusters was purposefully set to an ”incorrect” number to demonstrate that the number of cluster of a dataset is not trivial to discover, even in such a simple example. In this synthetic dataset, the number of clusters is not clear due to the two superimposed Gaussians. The number of clusters is a common initialization parameter for clustering methods. When no prior information about the dataset is given, the number of clusters can be hard to discover.

Cluster analysis is a relevant technique across several domains ([3]):

- grouping users with similar behaviour or preferences in **customer segmentation**;



(a) Input data, unlabeled.

(b) Desired labels.

(c) K-Means's labels.

Figure 2.1: Gaussian mixture of 5 distributions. Fig. 2.1a shows the raw input data, i.e. how the algorithms ”sees” the data. Fig. 2.1b shows the desired labels for each point, which here means their corresponding Gaussian. Fig. 2.1c shows the output labels of the K-Means algorithm with the number of clusters (input parameter) set to 4.

- image segmentation in the field of **image processing**;
- clustering gene expression data, among other application, in the domain of **biological data analysis**;
- generation of hierarchical structure for easy access and retrieval of **information systems**;

Chapter 3

State of the art

This dissertation is concerned with the scalability and optimization of EAC to large datasets. The goal of this optimization is to expand the applicability of the EAC method to large datasets, which translates in optimizing both speed and space. To this end, several aspects were researched. Work related to clustering and the concept of big data is briefly reviewed in section 3.2.

Evidence Accumulation Clustering is a method of three parts. This dissertation is concerned with the scalability of the whole algorithm which means that different steps have to be optimized separately. This algorithm and existing work on its scalability is reviewed in section 3.1. The first step is the generation of an ensemble of clustering partitions. Increasing speed can be attained with either faster algorithms and/or faster computation of existing algorithms. Both these approaches to faster generation of the ensemble were pursued and researched. The former in the form of the still young field of quantum clustering, which is briefly reviewed in section 3.3, and the later in the field of parallel computation, more specifically on computation in GPUs, reviewed in section 3.4. While quantum clustering was only reviewed for the first step, the parallel computation paradigm was investigated with all steps in mind. For that reason, K-Means GPU versions were researched as well as a MST-based Single Linkage Clustering method.

3.1 Evidence Accumulation Clustering

3.1.1 Ensemble Clustering

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 (if using the L2 norm) [4] 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. The underlying idea behind ensemble clustering is to use multiple clusterings from one or more clustering algorithms and combine them in such a way that the final clustering is better than any of the individual ones.

Formulation

Some notation and nomenclature, adopted from [5], should be defined since it will be used throughout the remainder of the present work. The term *data* refers to a set X of n objects or patterns $X = \{\mathrm{x}_1, ..., \mathrm{x}_n\}$, and may be represented by $\chi = \{x_1, ..., x_n\}$, such that $x_i \in \mathbb{R}^d$. A clustering algorithm takes χ as input and returns k groups or *clusters* C of some part of the data, which form a *partition* P. A clustering *ensemble* $\mathbb{P}$ is group of N partitions. This means that:

$$\mathbb{P} = \{P^1, P^2, ...P^N\} \tag{3.1}$$

$$P^i = \{C_1^i, C_2^i, ...C_{k_i}^i\} \tag{3.2}$$

$$C_k^i = \{x_a, x_b, ..., x_z\} \tag{3.3}$$

where C_j^i is the $j-th$ cluster of the $i-th$ partition, which contains k_i clusters and n_j^i is the number of samples that constitutes that cluster, with

$$\sum_{j=1}^{k} n_j^i = n, \qquad i = 1, ..., N \tag{3.4}$$

3.1.2 Overview of Evidence Accumulation Clustering

The goal of EAC is then to find an optimal partition P^* containing k^* clusters, from the clustering ensemble $\mathbb{P}$. According to [5], P^* should have the following properties:

- **Consistency** with $\mathbb{P}$;
- **Robustness** to small variations in $\mathbb{P}$; 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. Since EAC is an unsupervised method, this typically will not be available and is used for validation purposes. EAC makes no assumption on the number of clusters in each data partition. Its approach is divided in 3 steps:

1. Produce a clustering ensemble $\mathbb{P}$ (the evidence);
2. Combine the evidence;
3. Recover natural clusters.

A clustering ensemble, according to [5], can be produced from (1) different data representations, e.g. choice of preprocessing, feature extraction, sampling; or (2) different partitions of the data, e.g. output of different algorithms, varying the initialization parameters on the same algorithm.

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 object 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 [5], revealing the underlying structure of the data. The construction of this matrix is at the very core of this method.

The co-association matrix itself doesn't output a clustering partition. Instead, it is used as input to other methods to obtain the final partition. Since this matrix is a similarity matrix it's appropriate to use in algorithms take this type of matrices as input, e.g. K-Medoids or hierarchical algorithms such as Single-Link or Average-Link. to name two. 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 $max\{\mathcal{C}\} - n_{ij}$.

The final clustering is done using the SL or AL algorithms. Each of this algorithms will take as input the co-associations 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 n should be such that it maximizes the cost of cutting the dendrogram from $n - 1$ clusters to n.

3.1.3 Examples of applications

EAC has been used with success in several applications:

- in the field of bioinformatics it was used for the automatic identification of chronic lymphocyt leukemia [6];

- also in bioinformatics it was used for the unsupervised analysis of ECG-based biometric database to highlight natural groups and gain further insight [7];

- in computer vision it was used as a solution to the problem of clustering of contour images (from hardware tools) [8].

3.1.4 Advantages and Disadvantages

An advantage of EAC is that it takes no input parameters such as the number of output clusters. Furthermore, it provides good accuracy on datasets with hard shapes [5]. On the other hand, a big disadvantage is space complexity of the evidence combination step. In this step, when using the standard EAC, a co-association matrix is build, resulting in a complexity of $O(N^2)$. This translates in an almost prohibitive use of this method for bigger datasets.

3.1.5 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.

p neighbors approach

The first approach, [5], proposes an alternative $n \times p$ co-association matrix, where only the p nearest neighbors of each sample are considered in the voting mechanism. This comes at the cost of having to keep track of the neighbors of each pattern in a separate data structure and also of pre-computing the p neighbors. The quadratic space complexity is then transformed to $O(2np)$ and usually $p < \frac{n}{2}$ (value for which both approaches 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 that used in [5]), 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). However, it should be noted that computing the p nearest neighbors requires the computation of n^2 distance values.

Increased sparsity approach

The second approach, presented in [9], 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 datasets, 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 datasets analysed in [9] 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.

Although the results appear promising, the present work aims to deal with datasets much larger than this and, as a consequence, this technique should be further evaluated and tested to attest to its usefulness to very large datasets.

3.2 Clustering with Big Data

When big data is in discussion, two perspectives should be taken into account.The first deals with the applications where data is too large to be stored efficiently. This is the problem that streaming algorithms such as [10] try to solve by analysing data as it is produced, close to real-time processing. The other perspective is big data that is actually stored and processed. The latter is the perspective the present work is concerned with.

Scalability of EAC within the big data paradigm is the concern of this work. Although this line of research hasn't been pursued before, cluster analysis of big data 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.

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 [3]. 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 to process or (3) parallelizing and distributing the computation. The solutions for each of this approaches are, respectively, one-pass algorithms, randomized techniques that reduce the input space complexity and parallel algorithms.

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. A cluster offers a parallelized and distributed solution, which is easier to scale.

According to [3], 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).

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. Literature suggests that quadratic (and even exponential in some cases) speedup may be achieved. Most of the approaches for such conversions make use of Groover's search algorithm, or a variant of it, e.g. [11]. 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 [12]Given that the scope of the thesis is to accelerate clustering, having the extra overhead of simulating the systems would not allow speedups.

The second approach is the computational intelligence 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. 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 are modeled after this concept, namely clustering for image segmentation [13, 14, 15], The other approach models data as a quantum system and uses the Schrödinger equation to evolve it.

In the following two sections these approaches for quantum inspired computational intelligence are explored.

3.3.1 The quantum bit approach

What is a 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 [16]. 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 as

$$[\psi] = \alpha[0] + \beta[1]$$

where ψ is an arbitrary state vector and α, β are the the probability amplitude coefficients of basis states $[0]$ and $[1]$, respectively. 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 $[000]$ 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 [17]. 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. To the purposes of the following sections, the concept of the oracle is more related to *oracles with internal randomness* [17].

3.3.2 Quantum K-Means

Several clustering algorithms [13, 14, 15], as well as optimization problems [18], are modeled after this concept. To test the potential of the algorithms under this paradigm, a quantum variant of the K-Means algorithm based on[14] was chosen as a case study.

Description of the algorithm

The Quantum K-Means (QK-Means) algorithm, as is described in [14], is based on the classical K-Means algorithm. It extends the basic K-Means with concepts from quantum mechanics (the qubit) and 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. [11] suggests that this operations may not be required to produce variability in the population of qubit strings. This is because, according to [19], 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.

3.3.3 Horn and Gottlieb's algorithm

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 and was later extended by Weinstein and Horn.

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 [20, 21]. 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 [21, 20, 22].

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) [20], 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 [20]. 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 [20].

Another method [21] 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. [23] parallelized this algorithm to the GPU obtaining speedups of up to two magnitudes relative to an optimized multicore CPU implementation.

3.4 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” [24]. 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 [25, 26, 27, 28], hierarchical clustering [29, 30], document clustering [31], image segmentation [32], integration in Hadoop clusters [33, 34], 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.

3.4.1 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 [35]. 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.

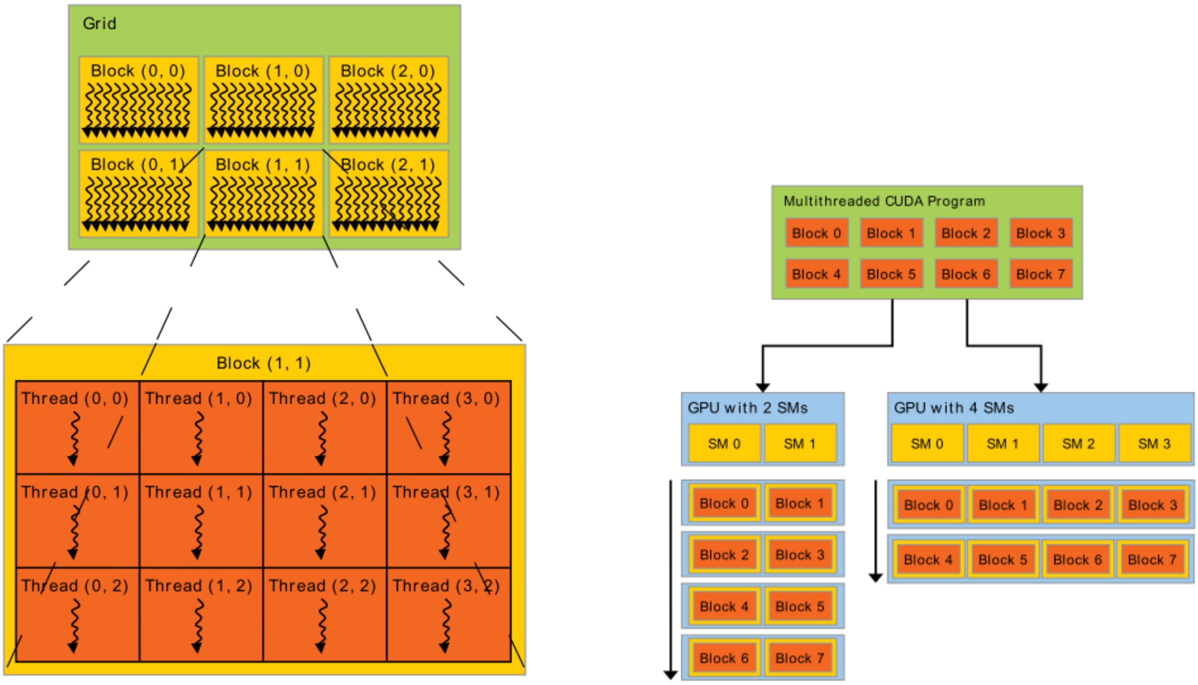
As Google's MapReduce computing model has increasingly become a standard for scalable and distributed computing over big data, attempts have been made to port the model to the GPU [36, 37, 38]. This translates in using the same programming model over a wide array of computing platforms.

3.4.2 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 [39, 40]. Literature also reports that performance from CUDA is better than OpenCL even for equivalent code.

3.4.3 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 [41], 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.



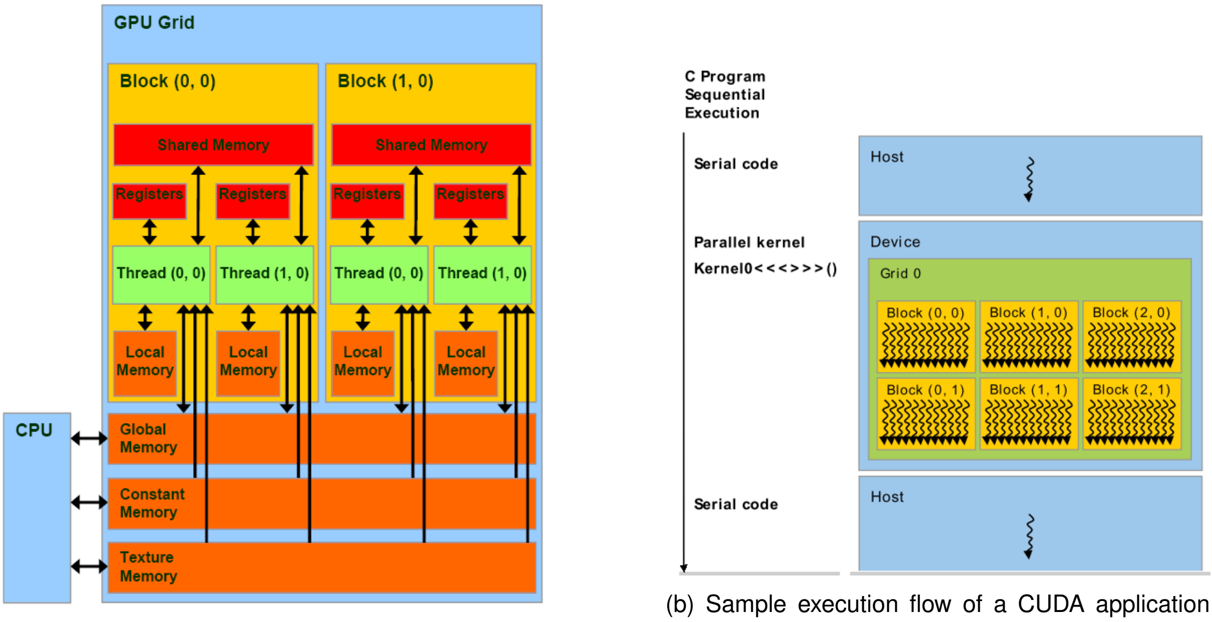
(a) Thread hierarchy [41].

(b) Automatic scalability [41].

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.

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.



(a) Memory model used by CUDA [41].

(b) Sample execution flow of a CUDA application [41].

Figure 3.2

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 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 Programming*. 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.

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 [42]. The results of aforementioned study showed datasets sufficiently large to make them relevant for the present work, considering the focus of the dissertation.

3.4.4 Parallel K-Means

K-Means is one of the building block of the EAC chain. Other algorithms can be used, but due to its simplicity and speed it is often used to produce ensembles varying the number of centroids to be used. Furthermore, it is a very good candidate for parallelization. Partitional clustering is an NP-hard problem and one of the most famous non-optimal solutions is the K-Means algorithm [43]. The sequential K-Means algorithm is composed by two main stages [2]:

1. **labeling stage** for the computation of the labels of each pattern in the dataset, e.g. the label of the $n-th$ pattern is 0 if the closest centroid is 0.
2. **update stage** for the recomputation of the centroids based on the labels assignment, i.e. the new centroids will be the mean of all the patterns assigned to it.

The K-Means algorithm is executed until a stopping condition is met, usually the number of iterations, a convergence criteria or both. The initial centroids are usually randomly chosen, but other schemes exist to improve the overall accuracy of the algorithm, e.g. K-Means++ [44]. After the first iteration, the first step is executed with the new centroids. The end result of the algorithm is the labels produced in the first step.

Several parallel implementations of this algorithm for the GPU exist [27, 25, 26, 32, 45]. The first step is inherently parallel as the computation of the label of the n-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 [25]. 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. This approach is suitable for devices with a low number of processors so as to stream the data to each one. The later approach is suited to devices with more processors. 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.

The approach taken in [46], 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 the computation). The parallelized algorithm's flow can be observed in Figure 3.3.

3.4.5 Parallel Single-Link Clustering

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 be clustered together.

The sequential SL algorithm works over a pair-wise similarity matrix and starts by considering that every data point is a separate cluster. Then, in each iteration, it selects the smallest weight that connects two clusters and merges the two clusters. The algorithm stops when $n-1$ merges have been performed,

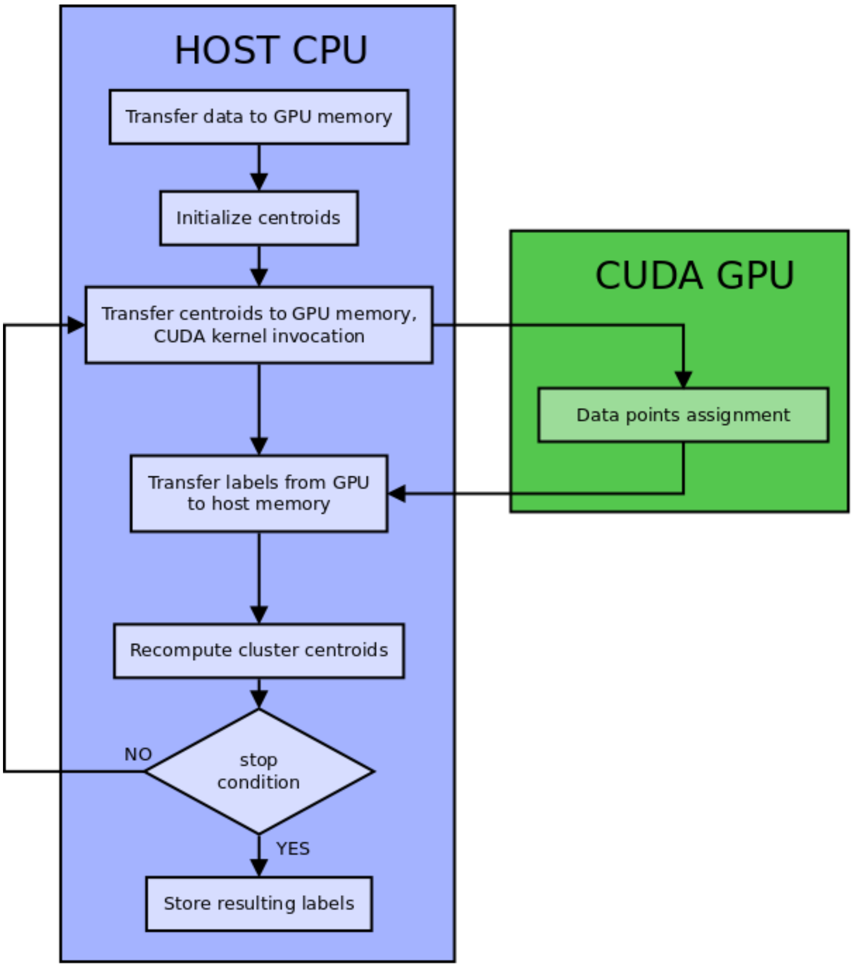

HOST CPU
Transfer data to GPU memory
Initialize centroids
Transfer centroids to GPU memory, CUDA kernel invocation
CUDA GPU
Data points assignment
Transfer labels from GPU to host memory
Recompute cluster centroids
NO
stop condition
YES
Store resulting labels

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

which is when all the data points have been connected in the same cluster. The output is a dendrogram connecting all the data points at different levels.

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 in EAC is computed with the part of the input (the co-association matrix) and, thus, not considered. An important relationship between SL and the Minimum Spanning Tree problem of graphs is the key to parallelize SL. If one takes the pair-wise similarity matrix to be a graph (each pattern is a node and each association an edge), then, when performing SL over this graph, the result can be interpreted as a structured MST. To get n clusters, one cuts the $n-1$ links with highest cost. If one takes this approach for solving the SL problem, it becomes easier to parallelize it since parallel MST algorithms are abundant in literature [47, 48, 49]. Furthermore, the MST approach to solve SL has been reported to be the fastest, since (1) it needs only $O(n)$ working space instead of the $O(n^2$ space of a pair-wise similarity matrix working copy and (2) it reads each similarity only once [50]. The same approach for extracting the final clustering in EAC was used in [51]. It should be noted that another algorithm with the characteristics mentioned for the MST approach for SL exist in another algorithm, the SLINK [52], which even has some advantages under certain conditions [50].

3.4.6 Algorithm for finding Minimum Spanning Trees

There are several algorithms for computing an MST. The most famous are Kruskal [53], Prim [54] and Borůvka [55]. 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 [48] and Prim's [**?**] algorithms exist, the focus will be on Borůvka's.

Several parallel implementations of this algorithm for the GPU exist, e.g. [47], [56] and [49]. Sousa et al. [49] 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 [49], 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.

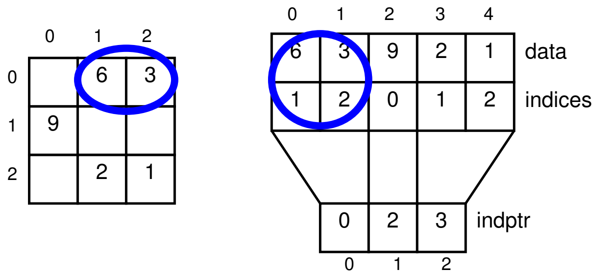


Figure 3.4: Correspondence between a sparse matrix and its CSR counterpart.

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:

- *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 3.4.6.

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.

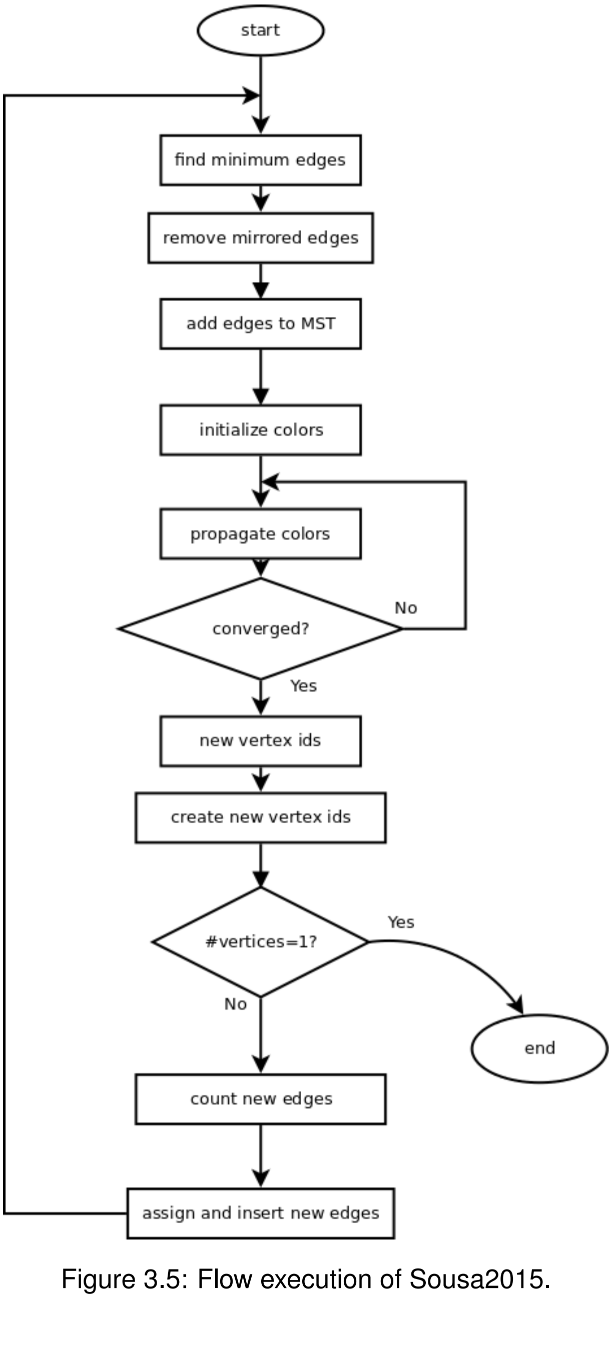


Figure 3.5: Flow execution of Sousa2015.

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 [49] 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 $[0, 1, 2, 3, 4, 5, 6, 7]$. 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 [57] and Blelloch's [58]. The two approaches focus on distinct efforts: the former focus on optimizing the number of steps [57] while the later focus on optimizing the amount of work done [58]. 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 [58] 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].

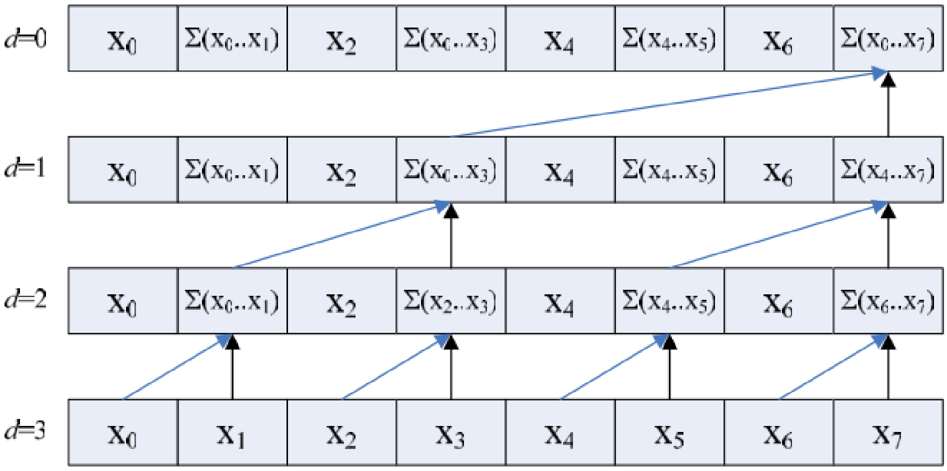


Figure 3.6: Representation of the reduce phase of Blelloch's algorithm [1]. *d* is the level of the tree and the input array can be observed at $d = 0$.

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).

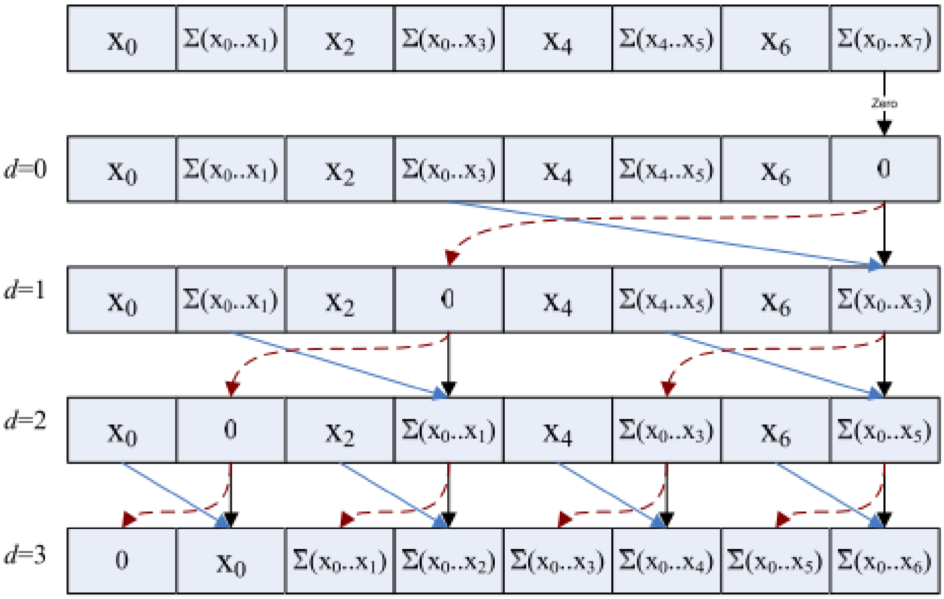

x0 | Σ(x0..x1) | x2 | Σ(x0..x3) | x4 | Σ(x4..x5) | x6 | Σ(x0..x7)
Zero
d=0
x0 | Σ(x0..x1) | x2 | Σ(x0..x3) | x4 | Σ(x4..x5) | x6 | 0
d=1
x0 | Σ(x0..x1) | x2 | 0 | x4 | Σ(x4..x5) | x6 | Σ(x0..x3)
d=2
x0 | 0 | x2 | Σ(x0..x1) | x4 | Σ(x0..x3) | x6 | Σ(x0..x5)
d=3
0 | x0 | Σ(x0..x1) | Σ(x0..x2) | Σ(x0..x3) | Σ(x0..x4) | Σ(x0..x5) | Σ(x0..x6)

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

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

Chapter 4

Methodology

The aim of this thesis is the optimization and scalability of EAC, with a focus for large datasets. EAC is divided in three steps and each has to be considered for optimization.

The first step is the accumulation of evidence, i.e. generating an ensemble of partitions. The main objective for the optimization of this step is speed. Using fast clustering methods for generating partitions is an obvious solution, as is the optimization of particular algorithms aiming for the same objective. Since each partition is independent from every other partition, parallel computing over a cluster of computing units would result in a fast ensemble generation. Using either or any combination of these strategies will guarantee a speedup.

Initial research was under the field of quantum clustering. After this pursuit proved fruitless regarding one of the main requirements (computational speed), the focus of researched shifted to parallel computing, more specifically GPGPU.

The second step is mostly bound by memory. The complete co-association matrix has a space complexity of $\mathcal{O}(n^2)$. Such complexity becomes prohibitive for big data, e.g. a dataset of 2×10^6 samples will result in a complete co-association matrix of $14901\ GB$ if values are stored in single floating-point precision. This work was focused on sparse matrices and cutting associations.

The last step has to take into account both memory and speed requirements. The final clustering must be able to produce good results and be fast while not exploding the already big space complexity from the co-association matrix. The work in this last step was initially directed towards parallelization and sparse matrices.

All the algorithms were implemented in Python with high reliance on numerical and scientific modules, namely NumPy [59] and SciPy [60, 61, 62]. Important modules for visualizing and processing statistical results were MatplotLib [63] and Pandas [64], respectively. The SciKit-Learn [65] machine learning library has a plethora of implemented algorithms which were used for benchmarking as well as integration in the devised solutions. The iPython [66] module was used for interactive computing which allowed for accelerated prototyping and testing of algorithms. Python is a interpreted language which translates in its performance being worse than a compiled language such as C or Java. To address this problem, the open-source Numba module from Continuum Analytics was used to allow for writing code in native Python and have it converted to fast compiled code. Furthermore, this module provides a pure Python interface for the CUDA API. The proprietary NumbaPro module was for two high level CUDA functions, namely *argsort* and *max*.

4.1 Quantum Clustering

Research under this paradigm aimed to find a solution for the first and last steps. Two venues were explored: Quantum K-Means and Horn and Gottlieb's quantum clustering algorithm. For both, the experiments that were setup had the goal of evaluating the speed and accuracy performances of the algorithms.

Under the qubit concept, no other algorithms were experimented with since the results for this particular algorithm showed that this kind of approach is infeasible due to the cost in computational speed. The results highlight that fact.

4.2 Speeding up Ensemble Generations with Parallel K-Means

K-Means is an obvious candidate for the generation of partitions since it is simple, fast and partitions don't require big accuracy - variability in the partitions is a desirable property which translates in few iterations. For that reason, optimizing this algorithm ensures that the accumulation of evidence is performed in an efficient manner. Furthermore, it is not necessary for K-Means to produce accurate clusterings, e.g. K-Means doesn't have to converge - 3 iterations should suffice. The reason for this is the desire for variability within the partition population.

4.3 Dealing with space complexity of coassocs

4.3.1 Exploiting the sparsity of the co-association matrix

Which method is more effective in very large datasets, however, would depend on the dataset. The sparsity maximization approach got very low densities for some datasets, close to 0.01 in some cases. This is already a big improvement

Either way, the CSR data structure is used to store the co-association matrix. Due to the sparse nature of the co-association matrix, storing it n this format can decrease used space to as much as 10%, depending on the sparsity of the matrix as shown in [9]. This is also an important step since the co-association matrix is already in the correct format for the computation of the final clustering within the GPGPU paradigm.

4.3.2 Using prototypes

k-Nearest Neighbors as prototypes

In the literature review, another approach to reduce the complexity of the co-association matrix is to use the p closest neighbors of each sample. Previous to building the co-association matrix, the p closest samples to each sample are computed and stored in the $n \times p$ neighbor matrix. The co-association matrix is reduced to size $n \times p$ and only considers the neighbors of each sample during the voting mechanism. This means two $n \times p$ matrices have to be stored, which, as long as p is significantly lowers than the number of samples, is close to a sparse representation of the full matrix.

It would be ideal to combine both approaches (sparsity and neighbors) to further reduce space complexity, but they're not necessarily compatible. When the neighbor approach is used, it is unlikely that a sample will never be clustered with it's closest p neighbors. However, this highly depends on the the number of neighbors relative to the number of samples and also the granularity of the partitions. If the number of neighbors is high, one of the neighbors can be sufficiently far away from the sample to not clustered with it. If the granularity of the partitions is high (i.e. there are a high number of small clusters) then each cluster may have sufficiently few samples that neighbors are not included. This means that the $n \times p$ co-association matrix may not have many zeros which translates in little return for using the sparsity augmentation approach. To illustrate this point, let's consider a dataset of 10^6 patterns.

Random prototypes

A second prototype approach is to choose p random non-repetitive samples as prototypes. This will be the same for every sample. Here the voting mechanism is altered so that if a sample is clustered with any of the prototypes, the correspondent element in the co-association matrix is incremented. This has the advantage that only a $n \times p$ matrix needs to be stored along with a p array for the prototypes. Furthermore, if p if high enough to provide a representative sample of the dataset the results can be as good as the full matrix

Medoid prototypes

This approach is similar to the random prototypes but, instead of choosing p random samples from the dataset, the prototypes will be the representatives of the dataset from another algorithm, e.g. K-Medoids, K-Means.

4.4 Building the sparse matrix

Building a non-sparse matrix is easy and fast since the memory for the whole matrix is allocated and accessing any position of the matrix is direct. When using sparse matrices, the former is not true. In the specific case of EAC, there is no way to know what is the number of associations the co-association matrix will have which means it's not possible to pre-allocate the memory to the correct size of the matrix.

This translates in allocating the memory gradually which may result in fragmentation, depending on the implementation.

Experiments showed that building the matrix with the recommended sparse structures for building the matrix saturated the memory and were very slow. Using a CSR matrix didn't saturate memory but was too slow. The core of the solution devised is the CSR sparse structure, which has the desired properties of having a low memory trace and allows for fast computations.

4.4.1 EAC CSR

The first step is making an initial assumption on the maximum number of associations *max_assocs* that each sample can have. A possible rule is

$$max\_assocs = 3 \times bgs,$$

where bgs is the biggest cluster size in the ensemble. The cluster size is a good metric for trying to predict the number of associations a sample will have since it is the limit of how many associations each sample will have in each partition. Furthermore, one would expect that the cluster neighbors of each sample will not vary significantly, i.e. the same neighbors will be clustered together with a sample repeatedly in many partitions. This scheme for building the matrix uses 4 supporting arrays:

- **indices** - an array of $n \times max\_assocs$ size that stores the columns of each non-zero value of the matrix, i.e. the destination sample to which each sample associates with;
- **data** - an array of $n \times max\_assocs$ size that stores all the non-zero values;
- **indptr** - an array of n size where the *i-th* element is the pointer to the first non-zero value in the *i-th* row.
- **degree** - an array of n size that stores the number of non-zero values of each row.

Each sample (row) has a maximum of *max_assocs* pre-allocated that it can use to fill with new associations. New associations that exceed this pre-allocated maximum are discarded. The *degree* array is the one keeping track of the number of associations of each sample.

Before continuing with the explanation of how the matrix is actually filled, it should be noted that this method assumes that the clusters received come sorted in a increasing order. The first partition is inserted in a special way. Since it is the first and the clusters are sorted, it's a matter of copying the cluster to the *indices* array in the positions of each sample, with the exclusion of the sample itself. This process can be observed clearly in Fig. 4.1. Because it's the fastest partition to be inserted, it's picked to be the one with the least amount of clusters (which translated in more samples per cluster) so that each sample gets the most amount of associations in the beginning (on average). This is only appliable if the whole ensemble is provided, otherwise there is no way to know what is the biggest cluster of the whole ensemble. This increases the probability that any new cluster will have more samples that correspond to already established associations. Because the clusters are sorted and only a copy of each cluster was made, it's not necessary to sort each row of the matrix by column index.

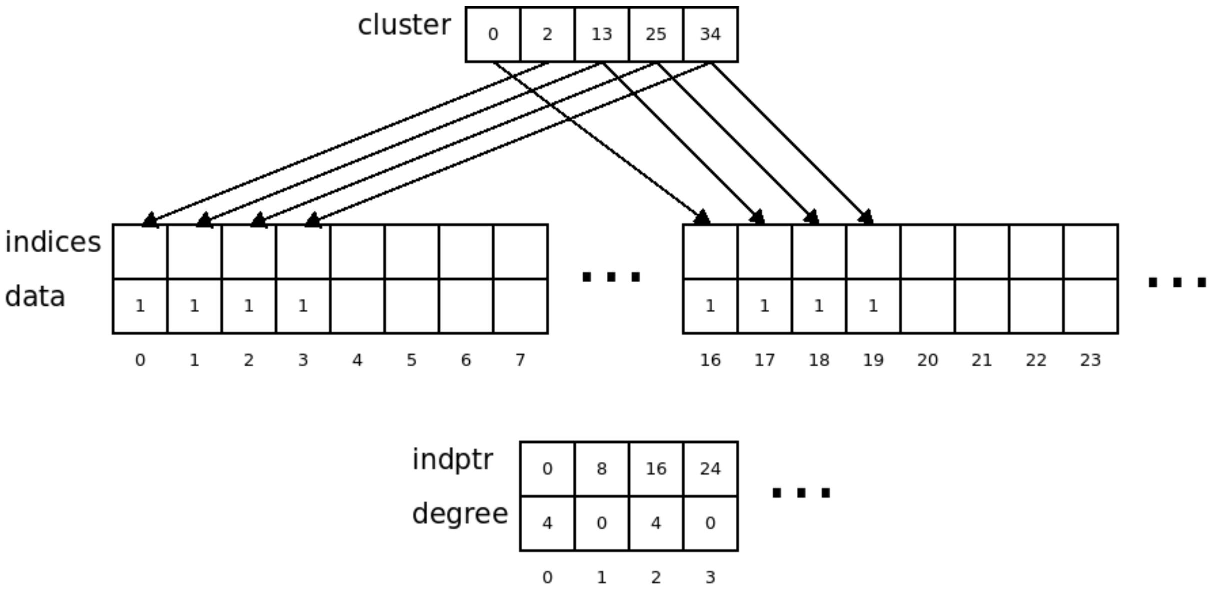


Figure 4.1: Inserting a cluster of the first partition in the co-association matrix.

For the remaining partitions, the process is different. Now, it's necessary to check if an association already exists. For each sample in a cluster it's necessary to add or increment the association to every other sample in the cluster. This is described in Algorithm 1.

Algorithm 1 Update matrix with cluster.

```
procedure UPDATE_CLUSTER($indices, data, indptr, degree, cluster, max\_assocs$)
    for $sample\ \mathbf{n}\ in\ cluster$ do
        for $every\ other\ sample\ \mathbf{na}\ in\ cluster$ do
            $ind = binary\_search(na, indices, interval\ of\ n\ in\ indices)$
            if $ind \geq 0$ then
                $increment\ data[ind]$
            else
                if $maximum\ assocs\ not\ reached$ then
                    $add\ new\ assoc.\ with\ weight\ 1$
```

Since a binary search is performed $(ns-1)^2$ times for each cluster, where ns is the number of samples in any given cluster, the indices of each sample must be in a sorted state at the beginning of each partition insertion. Two strategies for sorting were devised. The first strategy is to just insert the new associations at the end of the *indices* array (in the correct interval for each sample) and then, at the end of processing each partition, use a sorting algorithm to sort all the samples' intervals in the *indices* array. This was easily implemented with existing code, namely using *NumPy*'s *quicksort* implementation, which has an average time complexity of $O(n \log n)$

The second strategy is more involved. It results from the observation that, if one could know in which position each new association should be inserted, it would be possible to move all old associations to their final sorted positions and insert the new ones, all in an efficient manner with minimum comparisons (and thus branches in the execution of the code). For this end, the implemented binary search returns the index of the searched value (key) if it is found or $-ind - 1$, where ind is the position for a sorted insertion of the key in the array, if it is not found. This means that the position where each new association should be inserted is now available. Instead of just adding the new associations after the old ones of each sample, new associations are stored in two auxiliary arrays of size max_assocs: one for storing the samples of the associations *new_assoc_ids* and the other to store the indices that were the result of the binary search for each of the new associations *new_assoc_idx*, i.e. the indices where the new associations should be inserted. The process is illustrated with an example in Fig. 4.2, detailed in Algorithm 2 and explained in the following paragraph.

After each pass on a cluster (adding or incrementing all associations to a sample in the cluster), the new associations have to be added to the sample's indices interval in a sorted manner. The number of associations associatied with the i-th sample (*degree[i]*) is incremented by the amount of new associations to be added. An element is added to the end of the *new_assoc_idx* array with the value of *degree[i]* so that the last new association can be included in the general cycle. During the sorting process a pointer to the current index to add associations o_ptr is kept (it's initialized to the new total number of associations of a sample). The sorting mechanism looks at two consecutive elements in the *new_assoc_idx* array, starting from the end. If the i-th element of the *new_assoc_idx* array is greater or equal than the $(i-1)$-th element, then all the associations in the *indices* array between them (including the first element) are copied to the end of the *indices* interval, i.e. they're shifted to the right by i positions. Then, or in case the comparison fails, the $(i-1)$-th element of the *new_assoc_ids* is copied to the *indices* array in the position specified by *o_ptr*.

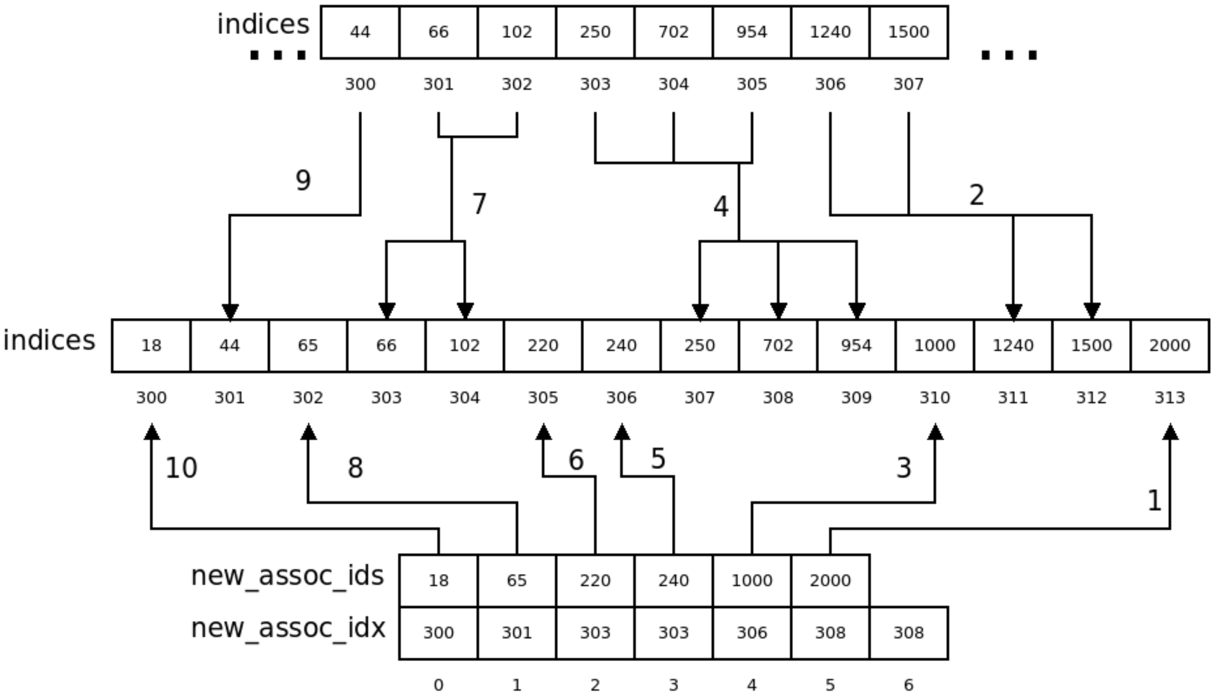


Figure 4.2: Inserting a cluster from a partition in the co-association matrix. The arrows indicate to where the indices are moved. The numbers indicate the order of the operation.

Algorithm 2 Sort the *indices* array in the interval of a sample n.

```
procedure SORT_INDICES($indices, data, indptr, degree, n, new\_assocs\_ptr, new\_assocs\_ids, new\_assocs\_idx$)
    $new\_assocs\_idx[new\_assocs\_ptr] = indptr[n] + degree[n]$
    $i = new\_assocs\_ptr$
    $o\_ptr = indptr[n] + new\_assocs\_ptr + degree[n] - 1$
    while $i \geq 1$ do
        $start\_idx = new\_assocs\_idx[i - 1]$
        $end\_idx = new\_assocs\_idx[i] - 1$
        while $end\_idx >= start\_idx$ do
            $indices[o\_ptr] = indices[end\_idx]$
            $data[o\_ptr] = data[end\_idx]$
            $end\_idx- = 1$
            $o\_ptr- = 1$
        $indices[o\_ptr] = new\_assocs\_ids[i - 1]$
        $data[o\_ptr] = 1$
        $o\_ptr- = 1$
        $i- = 1$
```

4.4.2 EAC CSR Condensed

The co-association matrix is symmetric, which means that only half is necessary to describe the whole matrix. That fact has consequences on both memory usage and computational effort on building the matrix. In the case of the fully allocated matrix, this translates in a reduction of approximately 50% of the memory required. Furthermore, only half the operations are made since only half the matrix is accessed, which also accelerates the building of the matrix. There is, however, a small overhead for translating the two dimensional index to the 1 dimensional index of the upper triangle matrix. When using a sparse matrix according to the EAC CSR scheme described above, there is no direct memory usage reduction. However, the number of binary searches performed by inserting new associations is half which has a significant impact on the computation time relative to the complete matrix.

Even though there is no direct memory reduction for switching to the condensed matrix form, this scheme does open possibilities for it. Previously, the number of associations for each sample (number of non zero values in each row) remained constant throughout the whole set of samples. However, using a condensed scheme means that the number of associations decreases the further down a sample is in the matrix, i.e. the closer to the bottom its respective row is. An example of this can be seen in the plot of the number of associations per sample in a complete and condensed matrix of Fig. 4.3. It's clear that, since the number of associations that is actually stored in the matrix decreases throughout the matrix, then the pre-allocated memory for each sample can decrease accordingly. One possible strategy, illustrated in Fig. 4.3, is to decrease the maximum number of associations linearly. In the example, the first 5% of samples have the 100% of the maximum number of associations and it decreases linearly until 5% of the maximum number of associations for the last sample.

Table 4.1 shows the memory usage of the different co-association matrix strategies in the general case and for an example of 100000 samples.

Table 4.1: Memory used for different matrix types for the generic case and a real example of 100000 samples. The relative reduction refers to the memory reduction relative to the type of matrix above, the absolute reduction refers to the reduction relative to the full complete matrix.

Type of matrix	Generic	Memory [MBytes]	Relative reduction	Absolute red.
Full complete	N^2	9536.7431	-	-
Full condensed	$\frac{N(N-1)}{2}$	4768.3238	0.4999	0.4999
Sparse constant	$N \times max\_assocs$	678.8253	0.1423	0.0711
Sparse condensed linear	$N \times max\_assocs$	372.8497	0.5492	0.0390

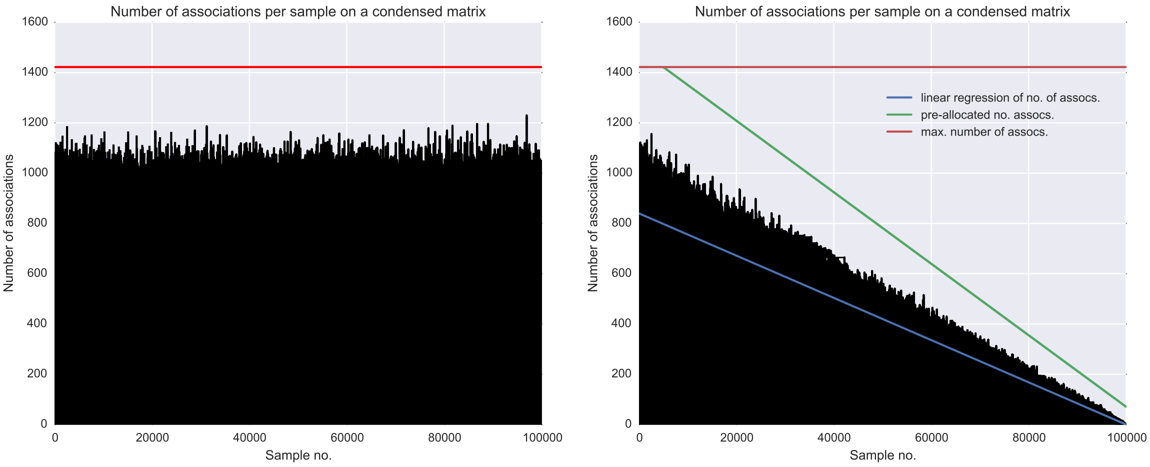


Figure 4.3: The left figure shows the number of associations per sample in a complete matrix; the right figure shows the number of associations per sample in a condensed matrix.

4.5 Hierarchical Agglomerative Clustering step

4.5.1 HAC and GPGPU

Since SL-HAC is an inherently sequential algorithm, an efficient GPU version is hard to implement. However, SL-HAC can be performed by computing the MST and cutting edges until we have the desired number of clusters. Considerable speedups are reported in the literature for MST solvers in the GPGPU paradigm. The considered solution uses the efficient parallel variant of Borůvka's algorithm [49]. If the number of clusters is given the necessary cuttings are done on the MST. If not, the number of clusters is computed with the lifetime technique. A new graph in the CSR format is computed from the altered MST and is then fed to the labelling algorithm, which will provide the final clustering.

Generating the MST

The output of the Borůvka's algorithm is a list of the indices of the edges constituting the MST. These indices point to the *destination* of the original graph. The original algorithm in [49] assumes fully connected graphs, but this is not guaranteed in the EAC paradigm. In graph theory, given a connected graph $G = (V, E)$, there is a path between any $s, t \in V$. In an unconnected graph, this is not the case and unconnected subsets are called components, i.e. a connected component $C \subset V$ ensures that for each $u, v \in C$ there is a path between u and v. In the present implementation, the issue of unconnected graphs was solved in the first step (finding the minimum edge connected to each vertex or supervertex). If a vertex has no edges connected to it (an outdegree of 0 since all edges are duplicated to cover both directions) then it is marked as a mirrored edge. This means that the independent components will be marked as supervertices in all subsequent iterations of the algorithm. The overhead of having this unnecessary vertices is low since the number of independent components is typically low compared to the number of vertices in the the graph and since the processing of such vertices is very fast. As a consequence, the stopping criteria becomes the lack of edges connected to the remaining vertices, which is the same as saying that all elements of the *outdegree* array are zero. This condition can be checked as a secondary result of the computation of the new *first_edge* array. This step is performed by applying an exclusive prefix sum over the *outdegree*. The prefix sum was implemented in such a way that it returns the sum of all elements, which translates in very low overhead to check this condition. The final output is the MST array and the number of edges in the array. The later is necessary because the number of edges will be less than $|V| - 1$ when independent components exist, and also because the MST array is pre-allocated in the beginning of the algorithm when the number of edges is not yet known.

Number of clusters and cutting edges

The number of clusters can be automatically computed with the lifetime technique or be supplied. Either way, a list (*mst_weights*) with the weights of each edge of the MST is compiled and ordered. The list of edges is also ordered according to the order of the *mst_weights*. If the number of clusters k was given, the algorithm removes the $k - 1$ heaviest edges. If there are independent components inside the MST those are taken into consideration before removing any edges. If the number of clusters k is higher than the number of independent components the final number of clusters will be the number of independent components.

To compute the number of clusters (which results in a truly unsupervised method) the lifetimes are computed. In the traditional EAC, lifetimes are computed on the weights of the clusters by the order they're formed. With the MST, the lifetimes are computed over the ordered *mst_weights* array, which is equivalent. If there are independent components, an extra lifetime that will serve as the threshold between choosing the number of independent components as the number of clusters or looking into the lifetimes within the MST. This is because links between independent vertices are not included in the MST. For this reason, the lifetime for going from independent edges to the heaviest link in the MST (where the first cut would be performed) is computed separately. If the maximum lifetime within the MST is bigger than the threshold, than the number of clusters is computed as in traditional EAC plus the number of independent components. Otherwise, the independent components will be the final clusters. Most of this process can be done by the GPU. Library kernels were used to sort the array and compute the $argmax$, and a simple kernel was used to compute the lifetimes.

Building the new graph

The final MST (after performing the edge cuts, if any) is then converted into a new, reduced graph. The function responsible for this takes the original graph and the final selection of edges contituting the MST and, afterwards, produces a graph in CSR format. The function has to count the number of edges for each vertex, compute the origin vertex of each edge (the original *destination* array only contains the destination vertices) and, with that information, build the final graph. This process can be done by the GPU with simple mapping kernels.

Final clustering

The last step is computing the final clusters. Any cut in the MST translates in independent components in the constructed graph. This means that the problem of computing the clusters translates into a problem of finding independent connected components in a graph, a problem that usually goes by the name of Connected Component Labeling. To implement this part in the GPU, the aforementioned Borůvka algorithm was modified to output the an array *labels* of length $|V|$ such that the $i-th$ position contained the label of the $i-th$ vertex. To this effect, the flow of the algorithm was slightly altered as shown in Figure 4.4. The kernel dealing with the MST was removed and a kernel to update the labels at each iteration, shown in Algorithm 3, was implemented. In the first iteration of the algorithm the converged colors are copied to the labels array. In every iteration the kernel to update the labels takes in the propagated colors and the array with the new vertex IDs. For each vertex in the array, the kernel first takes in the the color of the current vertex and maps it to the new color (remember that the color is actually a vertex ID and that that vertex has had its color updated). Afterwards, the kernel maps the new color with the new vertex ID that color will take, to keep consistency with future iterations.

Algorithm 3 Update component labels kernel

```
procedure UPDATE_LABELS($vertex\_id, labels, colors, new\_vertex$)
    $curr\_color \leftarrow labels[vertex\_id]$
    $new\_color \leftarrow colors[curr\_color]$
    $new\_color\_id \leftarrow new\_vertex[new\_color]$
    $labels[vertex\_id] \leftarrow new\_color\_id$
```

Memory transfer with the GPU

The whole algorithm of computing the SL clustering has been implemented in the GPU with minimizing the memory utilization in mind. Transferring the initial graph is the most relevant memory transfer. It has to be transfered twice: first for computing the MST and then to build the processed MST graph. This happens because the initial device arrays used for the graph are deallocated to give space for the arrays of subsequent iterations of the MST algorithm. This implementation design had in mind memory consumption in mind and could easily be avoided with the cost of having to store the bigger initial graph for the entire duration of the MST computation, which might be worthwhile if the GPU memory is abundant. The final labels array is transfered back to the host in the end of computation, but its size is

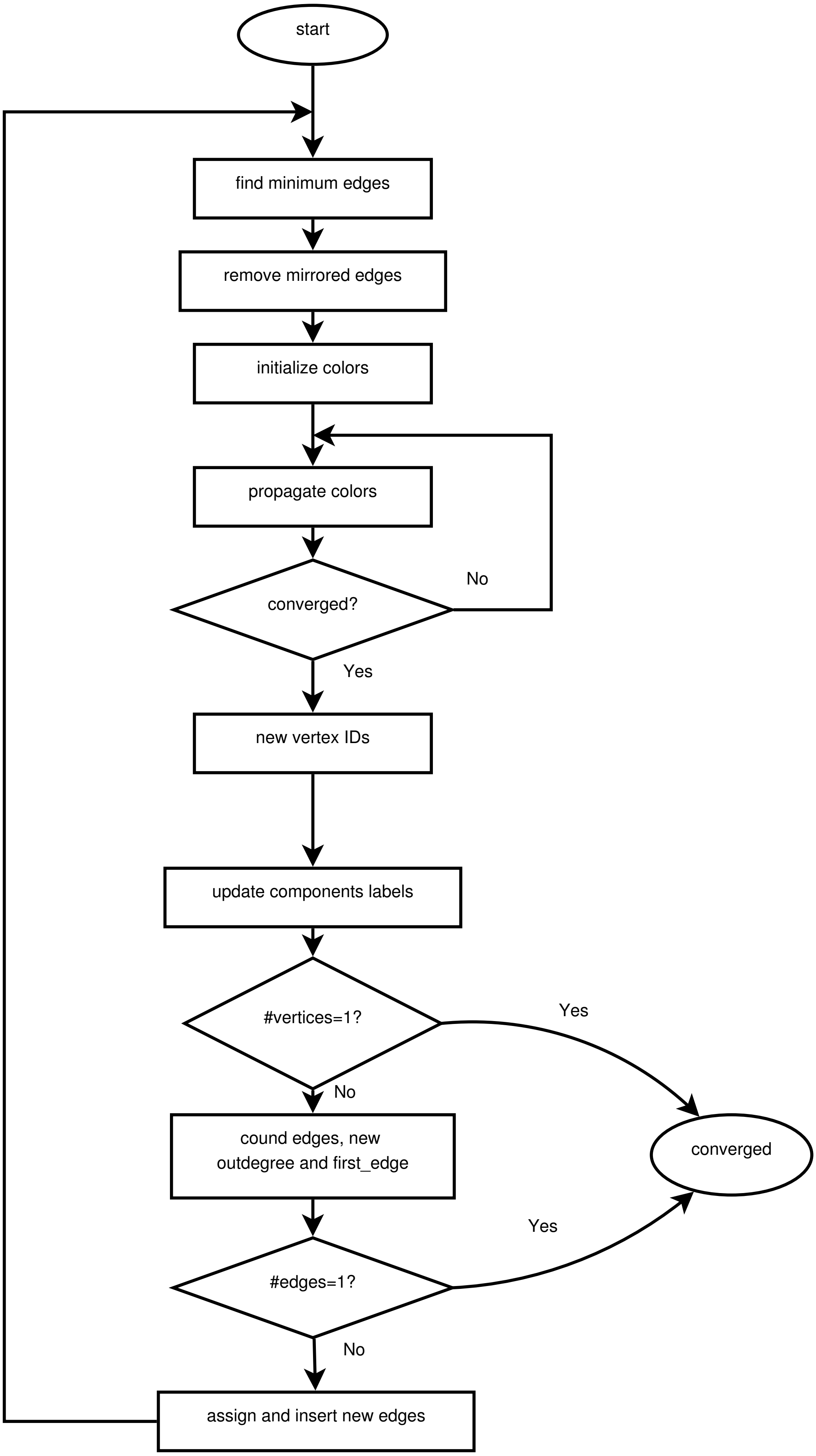


Figure 4.4: Diagram of the connected components labeling algorithm used.

small relative the to the size of the original graph. Furthermore, because the control logic is processed by the host and it is dependent on some values computed by the device, extra memory transfers of single values (e.g. number of vertices and number of edges on each iteration) are necessary. These, however, may be safely dismissed since they're of little consequence in the overall computation time.

	threshold	max_assocs	nnz percent relative to max
0	0.000000	1228.000000	0.979769
1	0.050000	794.000000	0.613603
2	0.100000	597.000000	0.471318
3	0.150000	496.000000	0.386988
4	0.200000	437.000000	0.325645
5	0.250000	367.000000	0.276588
6	0.300000	335.000000	0.235490
7	0.350000	294.000000	0.200629
8	0.400000	262.000000	0.170249
9	0.450000	232.000000	0.143376
10	0.500000	205.000000	0.119374
11	0.550000	167.000000	0.098120
12	0.600000	142.000000	0.079426
13	0.650000	121.000000	0.062757
14	0.700000	104.000000	0.048278
15	0.750000	88.000000	0.035731
16	0.800000	76.000000	0.024967
17	0.850000	68.000000	0.015932
18	0.900000	57.000000	0.008913
19	0.950000	38.000000	0.003906
20	0.000038	0.000086	0.000046

Chapter 5

Discussion?

5.1 real num assocs compared to samples per cluster

[9] reports that, on average, ¨the overall contribution of the clustering ensemble (including unbalanced clusters) duplicates the co-associations produced in a single balanced clustering with Kmin clusters. However, the spectrum of datasets evaluated regarding cardinality was smaller than that evaluated in the present work. The results suggest that the contribution of the ensemble is in fact higher.

5.2 Trade-off speed accuracy memory

When a problem of clustering of big data is at hand, the user should reflect upon what the problem at hand really requires: speed or accuracy. The user should take into consideration the nature of the data and the requirements of the problem (concerning speed and accuracy) before proceeding to the execution of the analysis. The present body of work reflects a method of clustering over big data using a high accuracy, but also high cost, method. Other methods offer the opposite, low cost, low to average accuracy.

5.3 GPU MST

The parallel version of the final step of EAC showed a slowdown relative to its sequential counterpart. This slowdown is related with the performance of the MST algorithm. The implementation of the algorithm was tested in some of the same graphs as those reported in [49] and revealed a speedup. However, when using the this MST solver on the target graphs (the co-association matrices) not only there was no speedup, but a significantly slowdown was observed, reaching up to nine times slower. The underlying reason for this is believed to be that the number of edges to node ratio of these graphs is low compared to that typically seen in co-associations matrix, even when using a prototype subset of the original matrix. Since the parallel computation is anchored to nodes, the workload per node is higher from the beginning and can increase significantly as the algorithm progresses. Furthermore, the workload can become highly unbalanced with some threads having to process hundreds of thousands of edges while others only a few thousands.

5.4 Expanding this work to other scalability paradigms

The present work focused on two main approaches for scalability: GPGPU and out-of-core solution. A current trend in computing of big data is cluster computing, which allows for distributed and parallel computing. For the sake of completeness it is interesting to discuss if the ideas related to the former two paradigms are transferable to the later.

The concept of GPGPU is one of parallelization. It is based on the fact that each computing thread in the GPU will execute instructions on a restricted subset of the entire dataset. This idea is easily transferable to cluster computing, as it is a core concept on both paradigms. For this reason, the generation of the ensemble is a step of EAC that can be very easily transfered to a computing cluster.

Chapter 6

Conclusions

Insert your chapter material here...

6.1 Achievements

The major achievements of the present work...

6.2 Future Work

The programming model used for the GPU was CUDA, used through a Python library called Numba. This library offers an interface to access most of CUDA's capabilities, but not all. One of those capabilities is Dynamic Parallelism. This offers the ability of having a host kernel call on other host kernel without intervention from the host. This translates in the possibility of moving the control logic in the Borůvka variant (and also its the Connected Components Labeling variant) to the device, effectevely removing the memory transfer of values related with the control logic.

Adaptation of the present implementation to OpenCL. This brings major benefits in respect to portability since OpenCL supports most devices. Moreover, OpenCL's performance is catching in on that of CUDA's and since it's programming model was based on CUDA, it should be straightforward for developers to make the switch.

Application of EAC to the MapReduce framework will further expand the possibilities of application of EAC.

Study the integration of other clustering algorithms within the the EAC toolchain.

A good follow-up of the present work is to study the relationship between several metrics (e.g. sparsity, accuracy, maximum number of co-associations) and the complexity of the dataset. Some metrics for describing the complexity of the dataset exist (Tin Kam Ho) and it would be interesting to profile several datasets of different complexities and structures and search for the former relationship. On a performance perspective it could prove useful to deduce better rules to set the maximum number of associations in the sparse matrix. On a accuracy perspective it would be interesting to see if there are types of datasets that simply are not a good fit for EAC while other are. It would also be interesting to relate complexity with the threshold cut-off.

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] Charu C Aggarwal and Chandan K Reddy. *Data clustering algorithms and applications*. ISBN 9781466558229.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] Ellis Casper, Chih-Cheng Hung, Edward Jung, and Ming Yang. A Quantum-Modeled K-Means Clustering Algorithm for Multi-band Image Segmentation. 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.

[15] 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.

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

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

[18] Huaixiao Wang, Jianyong Liu, Jun Zhi, and Chengqun Fu. The Improvement of Quantum Genetic Algorithm and Its Application on Function Optimization. 2013(1), 2013.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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`.

[39] 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`.

[40] 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.

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

[42] 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`.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

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

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

[56] 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.

[57] 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.

[58] 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.

[59] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, 2011. ISSN 15219615. doi: 10.1109/MCSE.2011.37.

[60] Eric Jones, Travis Oliphant, Pearu Peterson, and Others. {SciPy}: Open source scientific tools for {Python}. URL http://www.scipy.org/.

[61] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, 2007. ISSN 15219615. doi: 10.1109/MCSE.2007.58.

[62] K. Jarrod Millman and Michael Aivazis. Python for scientists and engineers. *Computing in Science and Engineering*, 13(2):9–12, 2011. ISSN 15219615. doi: 10.1109/MCSE.2011.36.

[63] John D Hunter. Matplotlib: A 2D graphics environment. *Computing in science and engineering*, 9(3):90–95, 2007.

[64] Wes McKinney. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference*, 1697900(Scipy):51–56, 2010. URL http://conference.scipy.org/proceedings/scipy2010/mckinney.html.

[65] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL http://arxiv.org/abs/1201.0490.

[66] Fernando Pérez and Brian E. Granger. IPython: A system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, 2007. ISSN 15219615. doi: 10.1109/MCSE.2007.53.

Appendix A

Vector calculus

In case an appendix if deemed necessary, the document cannot exceed a total of 100 pages...

Some definitions and vector identities are listed in the section below.

A.1 Vector identities

$$\nabla \times (\nabla \phi) = 0 \tag{A.1}$$

$$\nabla \cdot (\nabla \times \mathbf{u}) = 0 \tag{A.2}$$