

Efficient Evidence Accumulation Clustering for large datasets/big data

Diogo Alexandre Oliveira Silva
Alferes Aluno Engenheiro Electrotécnico 136787-A

Thesis to obtain the Master of Science Degree in

Aeronautical Military Sciences in the speciality of Electrical Engineering

Examination Committee

Chairperson:	Professor Full Name
Supervisor:	Dra. Ana Luísa Nobre Fred
Co-supervisor:	Dra. Helena Aidos Lopes
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

Acknowledgments	v
Resumo	vii
Abstract	ix
List of Tables	xiii
List of Figures	xv
Glossary	xvii
1 State of the art	1
1.1 Scalability of EAC	1
1.1.1 p neighbors approach	2
1.1.2 Increased sparsity approach	2
1.2 Clustering with large datasets	3
1.2.1 General Purpose computing on Graphical Processing Units	4
1.2.2 Parallel K-Means	6
1.2.3 Parallel Single-Link Clustering	7
1.3 Quantum clustering	14
1.3.1 The quantum bit approach	15
1.3.2 Schrödinger's equation approach	18
Bibliography	27

List of Tables

List of Figures

1.1	Thread hierarchy [1].	6
1.2	Distribution of thread blocks is automatically scaled with the increase of the number of multiprocessors [1].	6
1.3	Memory model used by CUDA [1].	7
1.4	Sample execution flow of a CUDA application [1].	7
1.5	Flow execution of the GPU parallel K-Means algorithm.	8
1.6	Correspondence between a sparse matrix and its CSR counterpart.	10
1.7	Flow execution of Sousa2015.	12
1.8	Representation of the reduce phase of Blelloch's algorithm [2]. d is the level of the tree and the input array can be observed at $d = 0$	13
1.9	Representation of the down-sweep phase of Blelloch's algorithm [2]. d is the level of the tree.	14

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

State of the art

Scalability of EAC to large data sets is the concern of this work and, because of that, this chapter starts by reviewing what has been done in terms of scaling EAC in section 1.1. EAC is a method of three parts and this dissertation is concerned with the scalability of the whole algorithm which means that each step must be optimized. Scaling an algorithm means one has to take into account both speed of execution and memory requirements. Increasing speed can be attained with either faster algorithms and/or faster computation of existing algorithms. This chapter reflects research done within both approaches.

Although research on the application of EAC to large data sets has not been pursued before, cluster analysis of large data sets has. Since EAC uses traditional clustering algorithms (e.g. K-Means, Single-Link) in its approach, it is useful to understand how scalable the individual algorithms are as they will have a big impact in the scalability of EAC. Furthermore, valuable insights may be taken from the techniques used in the scalability of other algorithms. To this end, section 1.2 presents a brief review on cluster analysis of large data sets, with a focus on parallelization with GPUs. Furthermore, it offers a more detailed description of a GPU parallel version of K-Means and an approach for parallelizing Single-Link with the GPU.

An alternate approach on clustering for scaling with faster algorithms, the still young field of quantum clustering, was reviewed in section 1.3. This line of research was taken mostly with the first step of EAC in mind.

1.1 Scalability of EAC

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

1.1.1 p neighbors approach

The first approach, [3], proposes an alternative $n \times p$ co-association matrix, where only the p nearest neighbors of each pattern are considered in the evidence combination step. This comes at the cost of having to keep track of the neighbors of each pattern in a separate data structure of $O(np)$ memory complexity and also of pre-computing the p neighbors, which has a time complexity of $O(n^2)$ to compute the similarity measure from each pattern to every other pattern. The quadratic space complexity of the co-association matrix is then transformed to $O(2np)$: $O(np)$ for the actual co-association matrix and $O(np)$ for keeping track of the neighbors. Since usually one has $p < \frac{n}{2}$ (value for which both this approach and the original $n \times p$ matrix would take the same space), the cost of storing the extra data structure is lower than that of storing an $n \times n$ matrix, e.g. for a dataset with 10^6 patterns and $p = \sqrt{10^6}$ (a value much higher than the 20 neighbors used in [3]), 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).

1.1.2 Increased sparsity approach

The second approach, presented in [4], 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 latter 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 ($\{K_{min}, K_{max}\} = \{\frac{n}{A}, \frac{n}{B}\}, A < B$).

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

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

The number of samples of the data sets analysed in [4] 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 does not take into account extra data structures that accompany real sparse matrices implementations.

1.2 Clustering with large datasets

When large datasets, and big data, is in discussion, two perspectives should be taken into account [5]. The first deals with the applications where data is too large to be stored efficiently. This is the problem that streaming algorithms such as LOCALSEARCH [6] try to solve by analyzing data as it is produced, close to real-time processing. The other perspective is data that is actually stored for later processing which is the perspective relevant to the present work and will be further discussed below.

The flow of clustering algorithms typically involves some initialization step (e.g. choosing the number of centroids in K-Means) followed by an iterative process until some stopping criteria is met, where each iteration updates the clustering of the data [5]. In light of this, to speed up and/or scale up an algorithm, three approaches are available: (1) reduce the number of iterations, (2) reduce the number of patterns and/or features to process or (3) parallelizing and distributing the computation. The solutions for each of this approaches are, respectively, one-pass algorithms (e.g. CLARANS [7], BIRCH [8], CURE [9]), randomized techniques that reduce the input space complexity (e.g. PCA, CX/CUR [10]) and parallel algorithms (parallel K-Means [11], parallel spectral clustering [12]).

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 is harder to scale the above solutions for even bigger datasets. On the other hand, GPUs can be found on a large variety of computing platforms, from mobile devices to workstations and datacenters. A cluster offers a parallelized and distributed solution, which is easier to scale. According to [5], 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 than the direct access of the CPU or GPU to memory (host or device).

The present work is oriented towards GPU based parallelization, since GPUs are an easily accessible

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

1.2.1 General Purpose computing on Graphical Processing Units

The original intended use for GPUs was graphics processing, hence its name. Recently, GPUs have been increasingly used for other purposes - a trend commonly known as General Purpose processing in Graphics Processing Units (GPGPU). GPU present a solution for "extreme-scale, cost-effective, and power-efficient high performance computing" [13]. 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 the appearance of specialized GPU models designed for data computation (e.g. NVIDIA Tesla).

GPGPU application on several fields and algorithms has been reported with significant performance increase, e.g. application on the K-Means algorithm [14, 15, 16, 17], hierarchical clustering [18, 19], document clustering [20], image segmentation [21], integration in Hadoop clusters [22, 23], among other applications.

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

Programming GPUs

In the very beginning of GPGPU, programming was done directly through graphics APIs. Programming for GPUs was traditionally done within the paradigm of graphics processing, such as DirectX and OpenGL. If researchers and programmers wanted to tap into the computing power of a GPU they had to learn and use these APIs and frameworks, which is a challenging task since their general problems had to be modelled to the graphics-oriented primitives [24]. 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. Currently, the major programming models used for computation in GPU are OpenCL and CUDA. While the first is widely available in most devices, the latter 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 [25, 26, 27]. This translates in using the same programming model over a wide array of computing platforms.

OpenCL vs CUDA

Presently, the most mature programming models are CUDA and OpenCL. CUDA appeared first and is supported only by NVidia devices. It is also the most mature of the two and performs well since it was designed alongside with the hardware architecture of the supporting devices. OpenCL has the advantage of portability, but that comes with issues of performance portability. Both models are, in fact, very similar and literature suggests that porting the code from one to the other requires minimal changes [28, 29]. Literature also reports that CUDA performs better than OpenCL [29] for equivalent code.

Overview of CUDA

This section presents an overview of the CUDA programming model and its main concepts and characteristics. For a more thorough and extensive explanation of these topics, the CUDA C Programming Guides [1], the source of the present review, should be consulted. A GPU is comprised by one or several streaming processors (or multiprocessor). Each of these 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 divides the threads from a block into smaller batches, called *warps*. This hierarchy is represented in Figure 1.1. 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 1.2.

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 is 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, enabling inter-thread communication inside a block. Finally, each thread has access to local memory. Local memory resides in global memory space and has the same latency for read and write operations. However, if the thread is using only single variables or constant sized arrays, it uses register space to, which is very fast. If the memory used exceeds the available register space the local memory is used. This memory hierarchy is

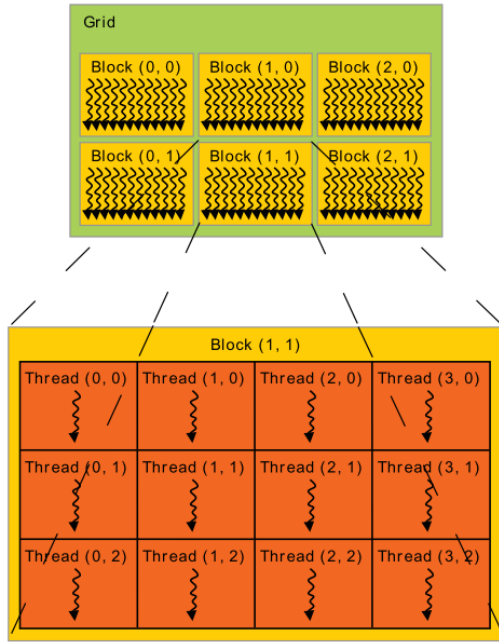


Figure 1.1: Thread hierarchy [1].

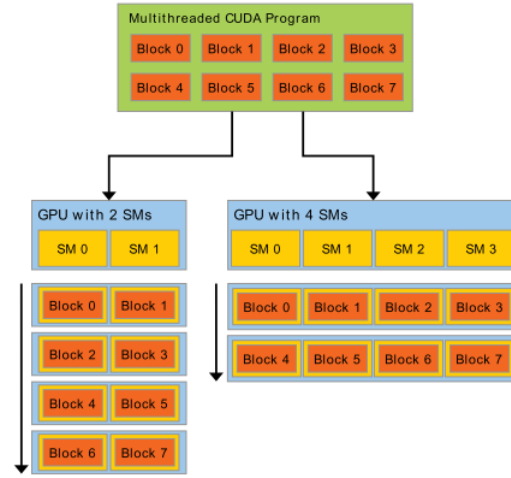


Figure 1.2: Distribution of thread blocks is automatically scaled with the increase of the number of multiprocessors [1].

represented in Figure 1.3.

The typical flow of a CUDA application (and, typically, any modern GPU application) is explained in this paragraph and can be observed in Figure 1.4. First, the host CPU is responsible for several steps in the set-up phase. The host starts by transferring any necessary to the device memory (global, texture or constant). The next step is 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 set-up phase is followed by the computation phase in the GPU. Finally, the host will transfer back the results from the device. It should be noted that the latest architectures support *Dynamic Parallelism*. This functionality allows the device to start other kernels without the intervention of the host CPU, which would alter the typical execution flow explained above if used. It can be particularly useful if several kernels have to be executed in an application from the same input data but with dependencies. In such a scenario, a block of the second kernel could be executed as soon as all dependencies from the first kernel were met, effectively cutting overheads for kernel calling from the host CPU.

1.2.2 Parallel K-Means

K-Means is an obvious candidate to generate the ensemble of the first step of EAC because it uses different initializations and parameters and due to its simplicity. Besides, K-Means is a very good candidate for parallelization. Still, other algorithms can be used to produce ensembles.

Several parallel implementations of this algorithm for the GPU exist [14, 15, 16, 21, 30] and all report significant speed-ups relative to their sequential counterparts in certain conditions, usually after the input data set goes above a certain cardinality, dimensionality or number of clusters threshold. The first step is inherently parallel as the computation of the label of the i -th pattern is not dependent on any other

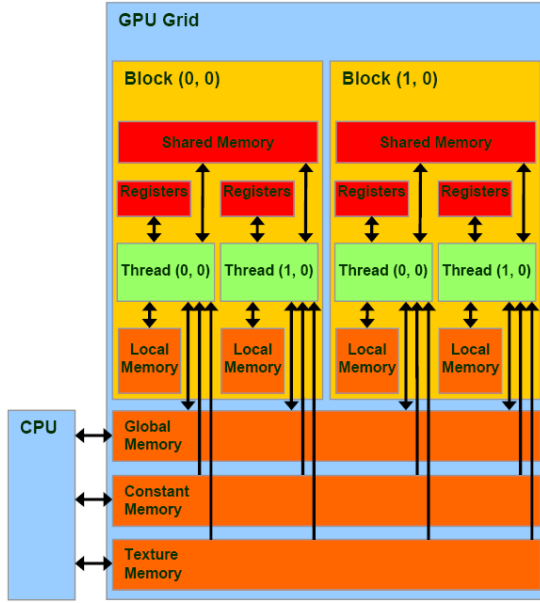


Figure 1.3: Memory model used by CUDA [1].

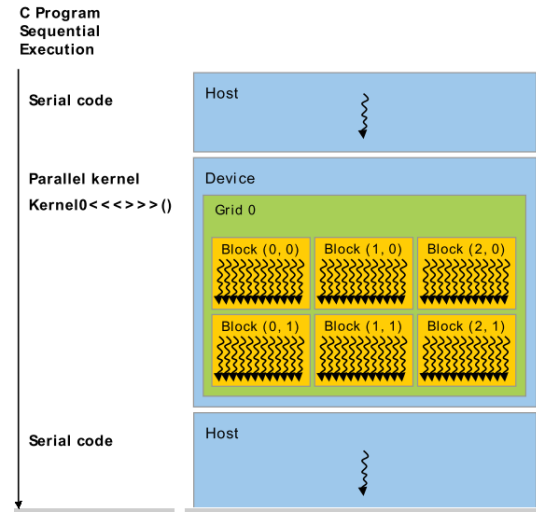


Figure 1.4: Sample execution flow of a CUDA application [1].

pattern, but only on the centroids. Two approaches to parallelize this step on the GPU are possible, a centroid-centric or a data-centric [14]. 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 to the closest centroid. In the latter, each thread will compute the distance from one or more data points to every centroid and determines to which centroid they are closest. This strategy has the advantage of using less memory since it does not need to store all the pair-wise distances to perform the labeling - it only needs to store the best distance for each pattern. According to [14], the former approach is suitable for devices with a low number of processors so as to stream the data to each one, while the latter is better suited to devices with more processors.

The approach taken in [11] 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 500 000 points. Furthermore, it should be noted that the speed up was measured against a sequential version with all C++ compiler optimizations turned on, including vector operations (which, by themselves, are a way of parallelizing computation). The parallelized algorithm's flow can be observed in Figure 1.5.

It should be noted that the literature reports that the performance of K-Means using Dynamic Parallelism is slightly worse than its standard GPU counterpart [31].

1.2.3 Parallel Single-Link Clustering

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

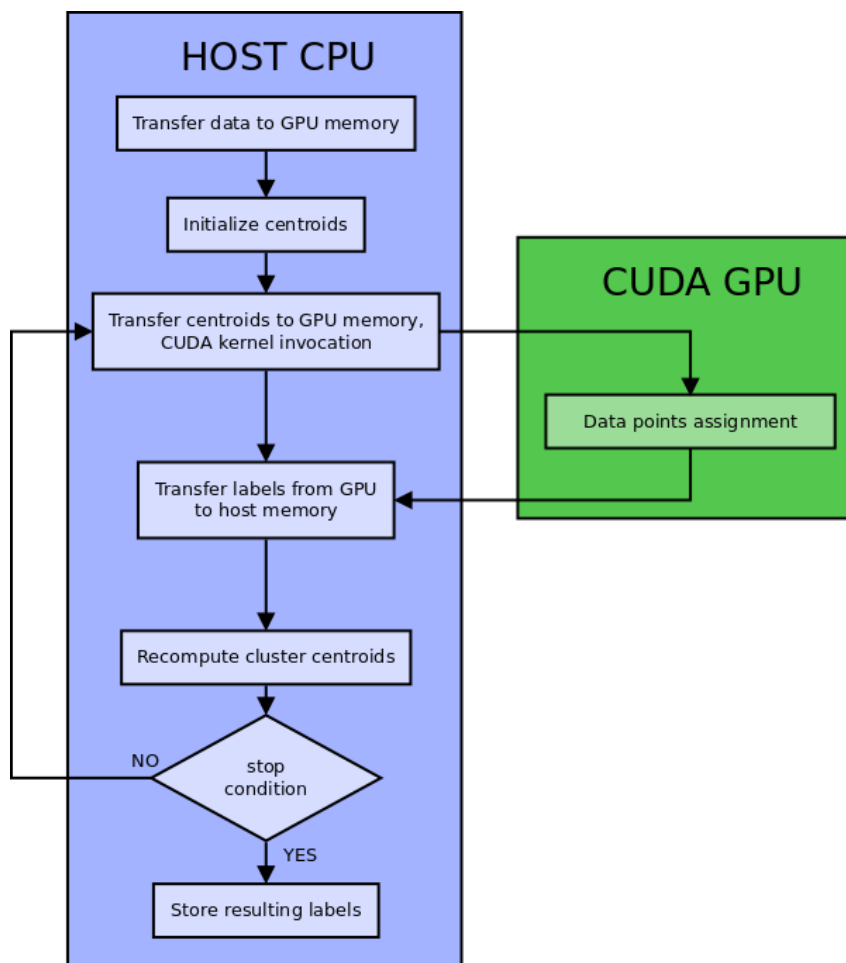


Figure 1.5: Flow execution of the GPU parallel K-Means algorithm.

SL is not easily parallelized since a new cluster generated at each step may include the one generated in the previous iteration. The most parallelizable part is the computation of the pair-wise similarity matrix, which is only useful if the input is raw data instead of a similarity matrix as in the case of EAC. The relationship between SL and the Minimum Spanning Tree, explained in chapter ??, is the key to parallelize it. If one takes this approach for solving the SL problem, it becomes easier to parallelize it since parallel MST algorithms are abundant in literature [32, 33, 34]. The same approach for extracting the final clustering in EAC was used in [35].

Algorithm for finding Minimum Spanning Trees

There are several algorithms for computing an MST. The most famous are Kruskal [36], Prim [37] and Borůvka [38]. Borůvka's algorithm is also known as Sollin's algorithm. The first two are mostly sequential, while the latter has the highest potential for parallelization, specially in the first iterations. As such, even though GPU parallel variants of Kruskal's [33] and Prim's [39] algorithms exist, the focus will be on Borůvka's.

Several parallel implementations of this algorithm for the GPU exist, e.g. [32], [40] and [34]. Sousa et al. [34] 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. This section will review the algorithm proposed in [34], 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. 1.6. It requires three arrays to fully describe the graph:

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

Within the algorithm's context, these three arrays are denominated as *first_edge*, *destination* and *weight*, respectively. There change in denomination is for making their purposes clearer. 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 are:

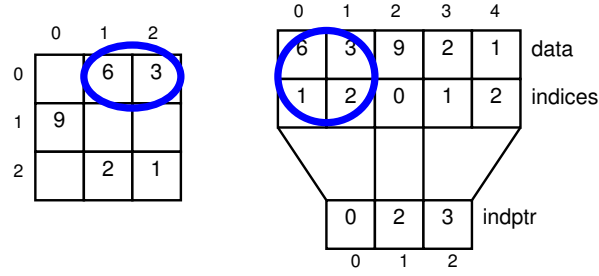


Figure 1.6: Correspondence between a sparse matrix and its CSR counterpart.

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

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

Steps of the algorithm

Within the context of the algorithm the *id* of a vertex is its index in the *first_edge* array, the *color* is related to the *ids* and the *successor* of a vertex is the destination vertex of one of its edges. The algorithm's flow is represented in Figure 1.7 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 will 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 1.2.3.
5. *Count, assign, and insert new edges*: the final step is where the final operations for building the contracted graph are performed. The algorithm will count the number of edges that each super-vertex has connecting to other super-vertices, i.e. the connections between subgraphs. This is simply accomplished by selecting every edge whose origin and destination colors are distinct. For each such edge the corresponding positions of the origin and destination super-vertices in a new *outdegree* array are incremented with an atomic operation. The new *first_edge* array is obtained from performing an exclusive scan over *outdegree*. The next step is to assign and insert edges in the contracted graph. Once again, the algorithm will determine which edges will be in the contracted graph by checking the origin and destination colors. A copy *top_edge* of the new *first_edge* array is done to keep track of where to insert the new edges. When an edge is assigned to a super-vertex it is inserted (in the new *weight* and *destination* arrays) in the position specified in *top_edge* and the algorithm increments *top_edge* so the next edge will not overwrite the previous one. The increment is done with an atomic operation since multiple edges can be assigned to the same super-vertex and the insertion of all edges is being done in parallel. It should be noted that duplicated edges are not 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 does not support unconnected graphs, i.e., it is not able to output a forest of MSTs. Upon contact, the author reported that a solution to that problem is, on the step of building the flag array, only mark a vertex if it is both the representative of its supervertex and has at least one neighbour.

Exclusive scan

The *scan* operation is one of the fundamental building block of parallel computing. Furthermore, two of the steps of the Borůvka variant of [34] are performed with an exclusive scan where the operation is a sum. To illustrate the functioning of the exclusive scan, let us 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 us further consider the input array to be the sequence

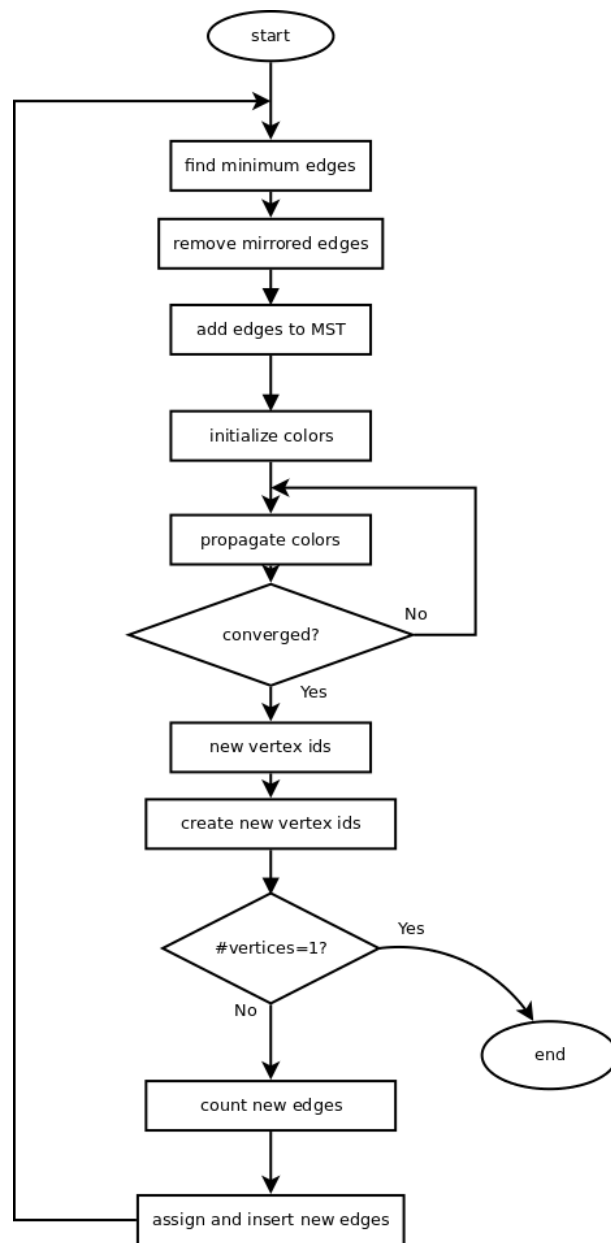


Figure 1.7: Flow execution of Sousa2015.

$[0, 1, 2, 3, 4, 5, 6, 7]$. Then the output will be $[0, 0, 1, 3, 5, 8, 13, 19]$. The first element of the output will be the identity (if it were an inclusive scan, it would be the first element itself). The second element is the sum between the first element of the input array and the first element of the output array, the third element is the sum between the second element of the input array with second element of the output array, and so on. This algorithm seems highly sequential in nature each element of the output array depends on the previous one, but two main parallel versions can be found in literature: Hillis and Steele's [41] and Blelloch's [42]. The two approaches focus on distinct efforts: the former focus on optimizing the number of steps [41] while the latter focus on optimizing the amount of work done [42]. 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* [2], 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 [2] or [42] for such details.

During the reduce phase (see Figure 1.8), 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 [2].

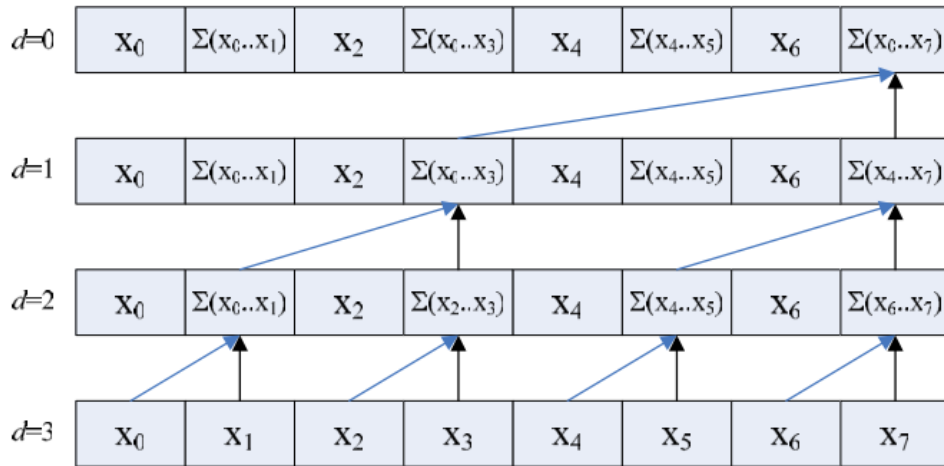


Figure 1.8: Representation of the reduce phase of Blelloch's algorithm [2]. d is the level of the tree and the input array can be observed at $d = 0$.

In the down-sweep phase (see Figure 1.9) the algorithm traverses back the tree. During the traversal, and using the partial sums calculated in the reduce phase, it builds the scan in place (overwriting the result of the reduce phase).

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

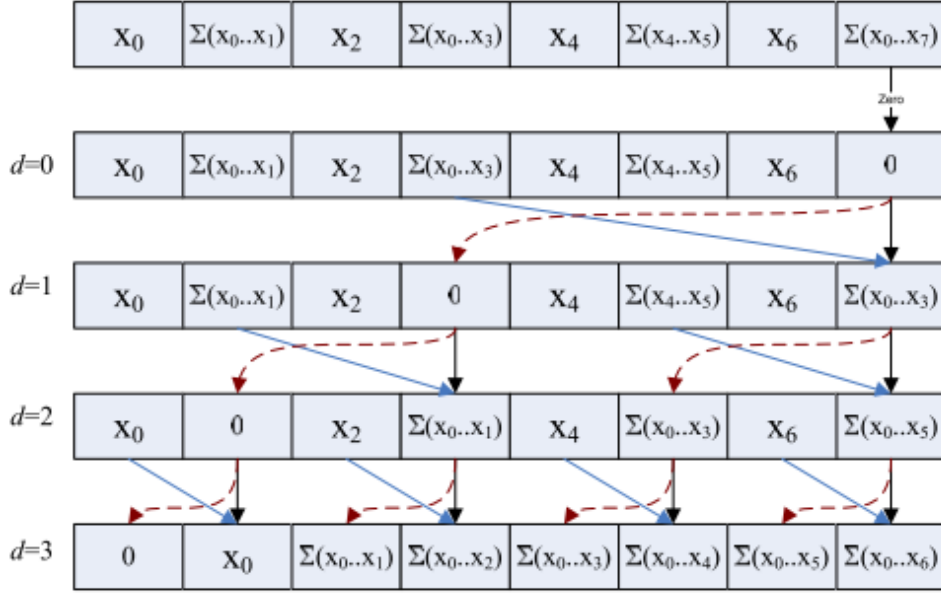


Figure 1.9: Representation of the down-sweep phase of Blelloch's algorithm [2]. d is the level of the tree.

However, [2] explains how to overcome this limitation and also offers an implementation for CUDA and hardware specific optimizations.

1.3 Quantum clustering

The field of quantum clustering has shown promising results regarding potential speed-ups in several tasks over their classical counterparts. Currently, two major approaches for the concept of quantum clustering were found in the literature: quantization of clustering methods to work in quantum computers or algorithms inspired by quantum physics.

The former approach translates in converting algorithms to work partially or totally on a different computing paradigm, with support of quantum circuits or quantum computers. Both Aïmeur et al. [43] and Lloyd et al. [44] show how the quantum paradigm can be used for speed-ups in the machine learning algorithms, with the possibility of obtaining exponential speed-ups. Many of these quantizations make use of Groover's database search algorithm [45], or a variant of it, e.g. Wiebe et al. [46]. Most literature on this approach is also mostly theoretical, since the physical requirements still do not exist for testing these methods. 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 that it is not feasible [47]. The quantization approach falls outside the scope of this dissertation, but Wittek [48] offers a thorough review on the state of the art of machine learning in the quantum paradigm.

The second approach is part of the wider field of Computational Intelligence. A study of the literature reveals that it typically further divides itself into two categories [49]. One comprises the algorithms based

on the concept of the qubit, the quantum analogue of a classical bit with interesting properties found in quantum objects. Several algorithms have been modeled after this concept, often also gathering inspiration from evolutionary genetic algorithms, which were successfully applied in several areas:

- tackling the Knapsack problem as described in Han and Kim [50] and its improved version [51];
- solving the traveling salesman problem [52];
- two implementations of a Quantum K-Means algorithm [53, 54];
- a Quantum Artificial Bee Colony algorithm [55, 49];
- two approaches for Fuzzy C-Means (FCM), namely Quantum New Weighted Fuzzy C-Means (QNW-FCM) [49] and Quantum Fuzzy C-Means (QFCM) [56, 49];
- a novel quantum inspired clustering technique based on manifold distance [57];
- a Quantum-inspired immune clonal clustering algorithm based on watershed [58].

The other approach models data as a quantum system and uses Schrödinger's equation in some way. Often, the data is modeled as a quantum system, where each pattern is a particle, and Schrödinger's equation is used to evolve the particle system into a solution. This has been applied in an optimization problem for electromagnetics using a quantum particle swarm [59] and also on several clustering algorithms:

- the Quantum Clustering [60] algorithm treats patterns as quantum particles whose potential is computed with Schrödinger's equation and the system is evolved with the Gradient Descent method;
- Dynamic Quantum Clustering [61] is a variation of Horn and Gottlieb [60] that uses Schrödinger's equation to evolve the system;
- a Fuzzy C-Means approach [62] based on Quantum Clustering [60];
- QPSO+FCM, a Fuzzy C-means [63] based on Quantum-behaved Particle Swarm Optimization (QPSO) [64].

For more information on quantum inspired algorithms, the reader is referred to Manju and Nigam [65] which offers a thorough survey on the state of the art of quantum inspired computation intelligence algorithms. The following two sections contain a brief overview of the concept of the qubit and the description of an algorithm that uses this approach. Afterwards, an algorithm that follows the Schrödinger's equation approach is reviewed.

1.3.1 The quantum bit approach

The quantum bit

To understand 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 Lanzagorta and Uhlmann [66]. 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 [53] as

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

where ψ is an arbitrary state vector and α, β are the probability amplitude coefficients of basis states $|0\rangle$ and $|1\rangle$, respectively. The Dirac bra ket notation is employed, where the *ket* $|\cdot\rangle$ corresponds to a column vector. The basis states correspond to the spin of the modeled particle (in this case, a fermion, 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:

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \end{bmatrix}$$

The probability of observing the state $|000\rangle$ will be $|\alpha_1|^2 \times |\alpha_2|^2 \times |\alpha_3|^2$. To use this model for computing purposes, black-box objects called *oracles* are used. Oracles are important to understand quantum speed-ups. They can be understood as subroutines that cannot be usefully examined or as unknown physical systems that perform a quantum operation on a qubit string [67] and with properties one would like to estimate. Within the context of the present work an oracle is an abstraction for the programmer. It is an object which can be called and changes state (which can be observed) as a consequence. For the purpose of the following sections, the concept of the oracle is more related to *oracles with internal randomness* [67] or, more simply, a probabilistic Turing machine, as in the case of [55].

Quantum K-Means

The Quantum K-Means (QK-Means) algorithm, as described in [53], is based on the classical K-Means algorithm. It extends the basic K-Means with concepts from quantum mechanics (the qubit) and evolutionary genetic algorithms.

Within the context of this algorithm, oracles contain strings of qubits and generate their own input by

observing the state of the qubits. After collapsing, the qubit value 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 are not 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 every qubit of each string in all oracles. 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. These oracles will have a higher probability of collapsing into initial centroid values closer to the best solution so far. This way, in future generations, the oracles do not initiate with the best centroids so far (which would not converge into a better solution) but we they are close while still ensuring diversity (which is also a desired property in 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. Literature suggests that this operations may not be required to produce variability in the population of qubit strings [51]. The reason for this is that enough variability is produced with the use of the angle-distance rotation method [51] in the quantum rotation operation, with a careful choice of the rotation angle. Still, when used, the goal of these operations is to produce further variability into the population of qubit strings.

1.3.2 Schrödinger's equation approach

The other approach to clustering that gathers inspiration from quantum mechanical concepts is to use the Schrödinger equation. The algorithm under study was created by Horn and Gottlieb [68] and was later extended by Weinstein and Horn [69].

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 [70, 69]. 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. Further details can be found in Weinstein and Horn [69], Horn and Gottlieb [70, 60].

With this information, one will compute the potential function $V(x)$ that corresponds to the state of minimum energy (ground state = eigenstate with minimum eigenvalue) [70], by solving the Schrödinger's equation in order of $V(x)$:

$$V(\mathbf{x}) = E + \frac{\sigma^2 \nabla^2 \psi}{\psi} = E - \frac{d}{2} + \frac{1}{2\sigma^2 \psi} \sum_{i=1}^N \|\mathbf{x} - \mathbf{x}_i\|^2 e^{-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}} \quad (1.1)$$

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

$$E = -\min_{\psi} \frac{\frac{\sigma^2}{2} \nabla^2 \psi}{\psi} \quad (1.2)$$

From equations 1.1 and 1.2, $V(x)$ can be computed. This potential function is related 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 [70]. However, it is very computationally intensive to compute $V(x)$ to the whole space, so the computation 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 these points toward some minima of the potential function. This is done with the gradient descent method in [70].

Another method [69] is to think of the input data as particles and use the Hamiltonian operator to evolve the quantum system in the time-dependent Schrödinger's 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. Wittek [71] parallelized this algorithm to the GPU obtaining speed-ups of up to two magnitudes relative to an optimized multicore CPU implementation.

Bibliography

- [1] Nvidia. CUDA C Programming Guide. *Programming Guides*, (August), 2015.
- [2] 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>.
- [3] 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.
- [4] 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.
- [5] Charu C Aggarwal and Chandan K Reddy. *Data clustering: algorithms and applications*. CRC Press, 2013. ISBN 9781466558229.
- [6] 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.
- [7] Raymond T Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *Knowledge and Data Engineering, IEEE Transactions on*, 14(5):1003–1016, 2002.
- [8] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD Record*, volume 25, pages 103–114. ACM, 1996.
- [9] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. In *ACM SIGMOD Record*, volume 27, pages 73–84. ACM, 1998.
- [10] Petros Drineas, Ravi Kannan, and Michael W Mahoney. Fast Monte Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition. *SIAM Journal on Computing*, 36(1):184–206, 2006.
- [11] 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.

- [12] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y Chang. Parallel spectral clustering in distributed systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(3):568–586, 2011.
- [13] 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> \delimeter"026E30F\$nh<http://dl.acm.org/citation.cfm?id=2287109>.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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>.
- [18] 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.
- [19] 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.
- [20] Yanjun Jiang, Enxing Li, and Zhanchun Gao. A GPU-based harmony k-means algorithm for document clustering. In *IET International Conference on Information Science and Control Engineering 2012 (ICISCE 2012)*, pages 3.29–3.29, 2012. ISBN 978-1-84919-641-3. doi: 10.1049/cp.2012.2426. URL <http://digital-library.theiet.org/content/conferences/10.1049/cp.2012.2426>.
- [21] J Sirotkovi, H Dujmi, and V Papi. K-Means Image Segmentation on Massively Parallel GPU Architecture. pages 489–494, 2012.

- [22] 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.
- [23] 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.
- [24] Marko J Miši, M Ć, and Milo V Tomaševi. Evolution and Trends in GPU Computing. *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 289–294, 2012.
- [25] 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.
- [26] 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.
- [27] 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>.
- [28] 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>.
- [29] 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.
- [30] 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>.
- [31] 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>.
- [32] 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.

- [33] 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.
- [34] Cristiano da Silva Sousa, Artur Mariano, and Alberto Proença. A Generic and Highly Efficient Parallel Variant of Boruvka’s Algorithm. 2015. URL <https://github.com/Beatgodes/BoruvkaUMinho>.
- [35] 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.
- [36] 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.
- [37] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.
- [38] Otakar Borůvka. O jistém problému minimálním. 1926.
- [39] Wei Wang, Yongzhong Huang, and Shaozhong Guo. Design and Implementation of GPU-Based Prim’s Algorithm. *International Journal of Modern Education and Computer Science*, 4(July): 55–62, 2011.
- [40] Pawan Harish, Vibhav Vineet, and P J Narayanan. Large graph algorithms for massively multi-threaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.
- [41] 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.
- [42] 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>.
- [43] Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. Quantum speed-up for unsupervised learning. *Machine Learning*, 90(February 2012):261–287, 2013. ISSN 08856125. doi: 10.1007/s10994-012-5316-5.
- [44] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411*, 2013. URL <http://arxiv.org/pdf/1307.0411.pdf>~~\$\delimiter"026E30F\$npapers2://publication/uuid/2BDCBB85-812C-46E4-B506-73B9A41EBBC1~~.
- [45] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

- [46] 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>.
- [47] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1982. ISSN 00207748. doi: 10.1007/BF02650179.
- [48] Peter Wittek. *Quantum Machine Learning: What Quantum Computing Means to Data Mining*. Academic Press, 2014.
- [49] 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.
- [50] Kuk-Hyun Han and Jong-Hwan Kim. Genetic quantum algorithm and its application to combinatorial optimization problem. *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, 2, 2000. doi: 10.1109/CEC.2000.870809.
- [51] 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.
- [52] H Talbi, A Draa, and M Batouche. A new quantum-inspired genetic algorithm for solving the travelling salesman problem. In *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on*, volume 3, pages 1192–1197 Vol. 3, 2004. doi: 10.1109/ICIT.2004.1490730.
- [53] Ellis Casper, Chih-Cheng Hung, Edward Jung, and Ming Yang. A Quantum-Modeled K-Means Clustering Algorithm for Multi-band Image Segmentation. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, volume 1, pages 158–163, 2012. URL http://delivery.acm.org/10.1145/2410000/2401639/p158-casper.pdf?ip=193.136.132.10&id=2401639&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.F7A57B2C5B227641.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=476955365&CFTOKEN=55494231&__acm__=1423057410_0d77d9b5028cb3.
- [54] 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.
- [55] Chih-Cheng Hung, Ellis Casper, Bor-chen Kuo, Wenping Liu, Edward Jung, Ming Yang, Xiaoyi Yu, Edward Jung, and Ming Yang. A Quantum-Modeled Artificial Bee Colony clustering algorithm for remotely sensed multi-band image segmentation. In *Geoscience and Remote Sensing Symposium (IGARSS), 2013 IEEE International*, pages 2501–2504. IEEE, 2013. ISBN 9781479911141.

- [56] Chih-Cheng Hung, Ellis Casper, Bor-Chen Kuo, Wenping Liu, Xiaoyi Yu, Edward Jung, and Ming Yang. A Quantum-Modeled Fuzzy C-Means clustering algorithm for remotely sensed multi-band image segmentation. In *Geoscience and Remote Sensing Symposium (IGARSS), 2013 IEEE International*, pages 2501–2504. IEEE, 2013.
- [57] Shenshen Liang, Cheng Wang, Ying Liu, and Liheng Jian. CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU. *Proceedings - 2009 IEEE Youth Conference on Information, Computing and Telecommunication, YC-ICT2009*, pages 415–418, 2009. doi: 10.1109/YCICT.2009.5382329.
- [58] Yangyang Li, Nana Wu, Jingjing Ma, and Licheng Jiao. Quantum-inspired immune clonal clustering algorithm based on watershed. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE, July 2010. ISBN 978-1-4244-6909-3. doi: 10.1109/CEC.2010.5586362. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5586362>.
- [59] Said M Mikki, Ahmed Kishk, and Others. Quantum particle swarm optimization for electromagnetics. *Antennas and Propagation, IEEE Transactions on*, 54(10):2764–2775, 2006.
- [60] 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>.
- [61] Marvin Weinstein and David Horn. Dynamic quantum clustering: A method for visual exploration of structures in data. *Physical Review E*, 80(6):066117, December 2009. ISSN 1539-3755. doi: 10.1103/PhysRevE.80.066117. URL <http://link.aps.org/doi/10.1103/PhysRevE.80.066117>.
- [62] Zhi-Hua Li, Shi-Tong Wang, and Jiangsu Wuxi. Quantum Theory: The unified framework for FCM and QC algorithm. In *Wavelet Analysis and Pattern Recognition, 2007. ICWAPR'07. International Conference on*, volume 3, pages 1045–1048. IEEE, 2007. ISBN 1424410665.
- [63] Hao Wang Hao Wang, Shiqin Yang Shiqin Yang, Wenbo Xu Wenbo Xu, and Jun Sun Jun Sun. Scalability of Hybrid Fuzzy C-Means Algorithm Based on Quantum-Behaved PSO. *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, 2(Fskd), 2007. doi: 10.1109/FSKD.2007.507.
- [64] Jun Sun, Bin Feng, and Wenbo Xu. Particle swarm optimization with particles having quantum behavior. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 325–331 Vol.1, 2004. doi: 10.1109/CEC.2004.1330875.
- [65] a. Manju and M. J. Nigam. Applications of quantum inspired computational intelligence: A survey. *Artificial Intelligence Review*, 42:79–156, 2014. ISSN 02692821. doi: 10.1007/s10462-012-9330-6.
- [66] Marco Lanzagorta and Jeffrey Uhlmann. *Quantum Computer Science*, volume 1. 2008. ISBN 9780511813870. doi: 10.2200/S00159ED1V01Y200810QMC002.

- [67] David Rosenbaum and Aram W. Harrow. Uselessness for an Oracle Model with Internal Randomness. *arXiv:1111.1462*, pages 1–23, 2011. ISSN 15337146.
- [68] David Horn, Tel Aviv, Assaf Gottlieb, Hod HaSharon, Inon Axel, and Ramat Gan. Method and Apparatus for Quantum Clustering, 2010.
- [69] 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>.
- [70] 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>.
- [71] 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>.

