UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER ON INFORMATION TECHNOLOGIES FOR BUSINESS INTELLIGENCE

Performance Analysis and Optimization of a Distributed Processing Framework for Data Mining Accelerated with Graphics Processing Units

---

IT4BI MSc THESIS

Student:
Edgar Isaac Hiroshi LEON SAIKI

Supervisors:
Dr. Toon CALDERS,
Nam-Luc TRAN,
Sabri SKHIRI

Barcelona, September 2014

A thesis presented by Edgar Isaac Hiroshi Leon Saiki
in partial fulfillment of the requirements for the MSc degree on
Information Technologies for Business Intelligence

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona

MASTER ON INFORMATION TECHNOLOGIES FOR BUSINESS INTELLIGENCE

Performance Analysis and Optimization of a Distributed Processing Framework for Data Mining Accelerated with Graphics Processing Units

by Edgar Isaac Hiroshi LEON SAIKI

In this age, a huge amount of data is generated every day by human interactions with services. Discovering the patterns of these data are very important to take business decisions. Due to the size of this data, it requires very high intensive computation power. Thus, many frameworks have been developed using Central Processing Units (CPU) implementations to perform this computation. For instance, a distributed and parallel programming model such as Google's MapReduce. On the other hand, since the last half decade, researchers have started using Graphics Processing Units (GPU) performance to process these huge data. Unlike CPU, GPU can execute many tasks in parallel. To measure the performance of GPU, EURA NOVA implemented two data mining algorithms (K-Means and Naive Bayes) in the framework to enable task execution in a distributed manner by considering availability of GPU power in each node. Even though the framework was successfully implemented, when compared to another CPU parallel framework, its performance was very poor. It shows that the framework does not use the performance of GPU effectively. Moreover, it contradicts with the fact that GPU can execute many tasks in parallel and thus, faster than CPU implementation. As a result, this research topic started with the objective to answer how to improve this performance. Specifically, to improve the performance of the K-Means implementation. We also included a new data mining implementation called Expectation Maximization to the framework, taking advantage of each GPU node and the distribution nodes. Furthermore, we address some good practices when implementing data mining in GPU from a sequential design.

Working with general purpose GPU is still in development stage. A well known library is Thrust. We used it to achieve the above objectives. Finally, we evaluated our solutions by comparing with other existed CPU frameworks. The results show that we improved the K-Means performance more than 130x, and plugged the expectation maximization implementation into EURA NOVA's framework.

Acknowledgements

I would like to express my deep gratitude to my supervisor Professor Toon Calders and EURA NOVA's Advisors: Nam-Luc Tran and Sabri Skhiri. For their patient guidance, enthusiastic encouragement and useful critiques of this research work.

I will always be in gratitude with Consejo Nacional de Ciencia y Tecnologia (CONACYT), the organization which granted me and gave me the opportunity to further continue my studies. As well as EURA NOVA, a young company founded in 2008, established in Belgium, where I did my master thesis.

I would also like to thank Dr. Jorge, Arnaud, Sahilu, Olivier and Alexander, for their advice and assistance during my master thesis. My grateful thanks are also extended to my IT4BI colleagues for their help during all the semesters as well as keeping me motivated.

My deepest appreciation to my parents, for always being supporting and caring for me during my entire life. Also, for encourage me throughout all my studies. Life could not have given me better parents, love you.

I would (particularly) like to thank Mitsue, without her, this would not had been possible, she is a role model to me. Also, I want to thank Sashiko and my aunt Chela for always cheering me up, love you all.

Finally, I am deeply grateful to Brenda, for all her support for so many years, as well as always being there when I needed her the most, love you.

Contents

List of Figures

List of Tables

To my family, specially to my brother Katzuo, miss you so much...

Chapter 1

INTRODUCTION

1.1 Context

Nowadays, lots of important data are generated in our daily life. These data are used by companies to take business decisions by using data mining techniques analyzing the meaning of this data [1]. As per pingdom website [14], data on the web is growing exponentially and there are more than two billion users generating data by interacting with services. Data mining is a field of computer science that plays a vital role in different areas of applications, like healthcare, market analysis, manufacturing, engineering, etc [15]. To enable the analysis of massive amounts of data in a reasonable time, parallel computing has been used and is getting huge attention since the past decade. Furthermore, in the past years we have seen the development of general purpose computing on Graphics Processing Units (GPU) [1], since GPUs [16] can theoretically deliver more parallel computing performance than traditional CPUs [17].

In GPU programming, there are different programming models which depend on the type of GPU device. For instance, the most commonly used programming models are CUDA [18] and OPENCL [19]. CUDA can only be used by Nvidia [18] GPU devices, whereas OPENCL can be used in any type of GPU Devices. On the other hand, it is also necessary to deal with memory allocation and memory access patterns [1]. There are some libraries that can deal with such low-level hardware considerations. For example, Thrust [20] is a library that contains parallel algorithms [21]. Thrust also presents good practices for using the GPU in general purpose computing [1].

There are two types of parallelism presented in EURA NOVA's implementation:

- Per node GPU, allowing further parallelism within each of the nodes.
- Multiple nodes, each equipped with its own CPU and Memory and connected with a network.

Per node GPU, where each node equipped with a GPU can execute its routine in parallel. Multiple nodes, where the program can be executed in the cluster to run in parallel. These two together add a second level of parallelization, making the performance of the computation faster [1]. SNIFF [1] is an appropriate framework for data mining that exploits both types of parallelism simultaneously. SNIFF was developed by EURA NOVA in order to prove that these two types of parallelism can achieve better computation for the GPU than its counterpart CPU.

In order to validate the framework and the execution model, Tran et al. [1] developed parallel GPU and CPU versions of two data mining algorithms in SNIFF: the Naive Bayes classifier [22] and the K-Means clustering algorithm [23]. These two algorithms were implemented in a distributed cluster system exploiting the massive parallel computation provided by the GPU in each node. The SNIFF framework resulted in a performance gain for the two aforementioned data mining problems for the GPU implementation outperforming the CPU implementation. The results are presented in the following graph [1]:

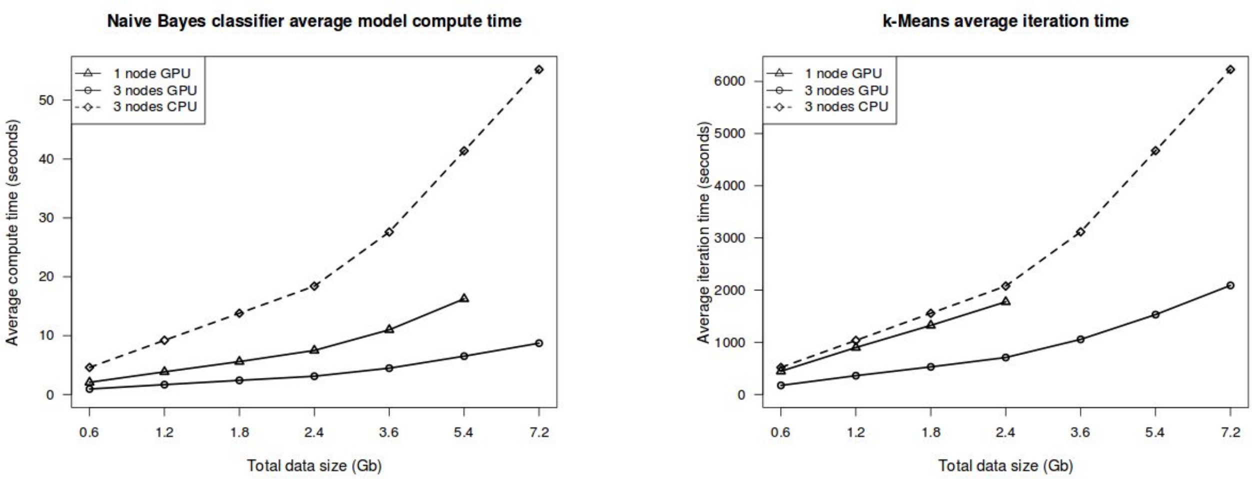

Figure 1.1: Average time spent on the computations of models. The figure on the left shows the average time spent on the computation of the model on 3 GPU nodes, on a single GPU node and on a CPU node for various sizes of the dataset for the Naive Bayes implementation. The figure on the right presents the average time spent on the computation of one K-Means iteration on a GPU node and on a CPU node for various sizes of the dataset [1].

Therefore, one can add more value and form a proper platform for distributed heterogeneous processing in the future by combining the best of the distributed processing models and the parallel processing capabilities of the GPUs [1].

The performance of K-Means GPU implementation in the SNIFF framework was tested with a Dataset of KDD Cup 1999 (Appendix A) with different sizes from 0.6 Gb to 7.2 Gb. The same dataset was applied to the CPU K-Means implementation of Mahout [24], which is another data mining framework exploiting parallel computation on a cluster.The starting point of this thesis was the observation that the results of SNIFF GPU K-means implementation were considerably slower than that of the Mahout CPU K-means, as can be seen in the following graph:

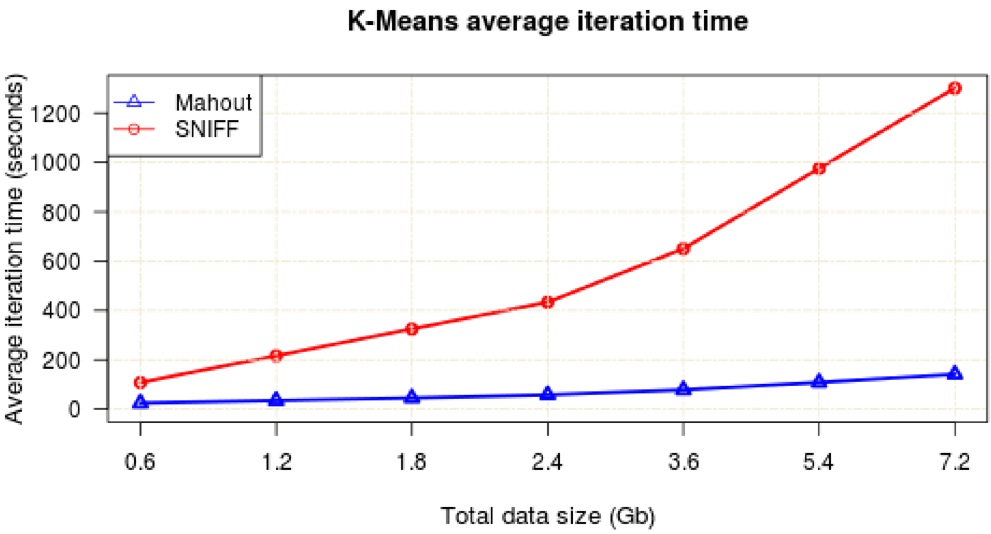

Figure 1.2: Average time spent for the computation on a GPU node Titan (Nvidia Graphics card [2]) and on the Mahout for one iteration time using various sizes of the dataset. Each of the nodes in the cluster locally processes one third of the total data size. In the case of Mahout, the data will be processed depending of how the data was distributed by Hadoop [3], using default block size of 64MB.

Theoretically, the GPU version has to be faster than the CPU version for computing tasks that require intensive computation. Therefore, the main goal of this project was to make the GPU implementation faster and find good practices to optimize the data mining implementations using GPU.

1.2 Motivation

Apart from the CPU, there are other different architectures that can be used to process the data, such as GPU [25]. GPU took their first steps as graphics accelerators, but nowadays, they are becoming more programmable and researchers started to use GPUs for non-graphics applications too [25]. Typically, graphics hardware contains a large number of programmable processors which are optimized to work efficiently, providing extremely high parallelism.

Nowadays, clusters with GPUs are being used throughout the world. Three of the five largest supercomputers in the world employ GPUs [25].

Researchers in data mining have demonstrated the superiority of the computational power of parallel processing in GPUs over that of CPU. For instance, in [26], the authors selected two clustering algorithms, the density-based algorithm DBSCAN [27] and the iterative algorithm K-Means. Both implementations using GPU outperformed their CPU counterparts. Other related works regarding K-Means implementations have shown significant speed ups as compared to their CPU counterparts [28–31]. Similar results were also obtained for K Nearest Neighbors implementation [32].

For EURA NOVA, the outcome of this thesis is not only important for optimizing the K-Means implementation for SNIFF and reviewing best practices in Data Mining using GPU. But also the insights in parallel programming obtained in the project could be used and add value in other projects of EURA NOVA, such as Arom [33]. The Arom project, which is based on a DataFlow Graph (DFG), can run distributed parallel data processing jobs on high volumes of data. The DFG model [1] is a graph where the operations are represented by nodes and data dependencies between operations are on the edges. Tran et al. [1] believe that an extension to the DFG model, e.g. Arom or Dryad [34], can be made using GPU, forming a suitable platform for distributed heterogeneous processing. As shown in the following figure [1]:

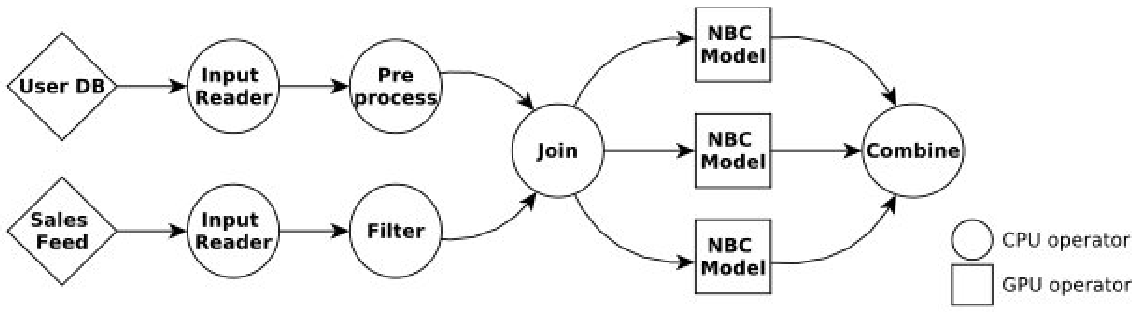

FIGURE 1.3: DFG representation of a job [1].

This DFG job contains two data sources, preprocessing stages and a join operation. Their results are the input to a GPU-enabled operation stage, represented as the squared nodes in the figure 1.3 [1].

1.3 Scope

The initial step of this research was to study the existing implementations of data mining algorithms from differents frameworks, such as Mahout [24], R [35] and Weka [36]. We mainly focused on the Mahout's K-Means implementation, since this implementation outperformed EURA NOVA's K-Means GPU implementation in an earlier study done by EURA NOVA. In the case of R and Weka, we focused in the sequential implementations to get some insights that may help to optimize the GPU K-Means implementation in SNIFF.

In addition, it was required to do a review of the conditions to efficiently execute data mining algorithms using GPUs. It was necessary to take into account some considerations when working with GPU, because a bad parallel implementation can lead to a terrible performance [1]. In this respect, Cristian Bohm et al. [26] proposed some data mining algorithms using GPUs. This review could help future works using data mining with GPU.

This study focused on optimizing a K-Means implementation in EURA NOVA's framework called SNIFF, as well as taking into account the differences of the optimized frameworks' implementations to integrate them in SNIFF. The study also included relevant information about best practices of Data Mining using GPU. As a result, the distributed processing framework for data mining with GPU built by EURA NOVA should have better time performance execution after optimizing the implementation.

1.4 Objectives

The first three goals of this master thesis according to [37] were:

1. To provide an analysis of the existing data mining algorithms and optimization of reference data mining frameworks such as Mahout, R and Weka.
2. To provide an in-depth review of the requirements for data mining algorithm to be optimally suited for GPUs.
3. To integrate the optimized algorithms into the prototype built by EURA NOVA.
4. As well as providing an implementation in the SNIFF framework of Expectation Maximization (EM) [38].

Regarding the implementation of the EM, we took it into consideration since EM is another algorithm that can be distributed and requires intensive computation. We will also give an insight of how this algorithm can fit the SNIFF framework using the capabilities for each node of GPU.

1.5 Initial Planning

The first step was to start by doing a "literature research" to study some of the algorithms of the frameworks previously mentioned. After doing this study, the following step was to start with an in-depth review of the requirements for the data mining algorithms to be suited for using a GPU. Consequently, all the optimizations will be implemented in the prototype already built by EURA NOVA [1]. After some tests, the execution time performance of the framework should be better, based on the optimizations of these implementations of the algorithms. If problems arise, then it may be necessary to review the implementations again. All the observations of the thesis should be written, making sure to understand the implementations of the algorithms of the open source frameworks and how these were optimized along with the requirements in order to use the best practices, so the GPU can work as efficient as possible.

Timeframe estimation for the Initial Planning:

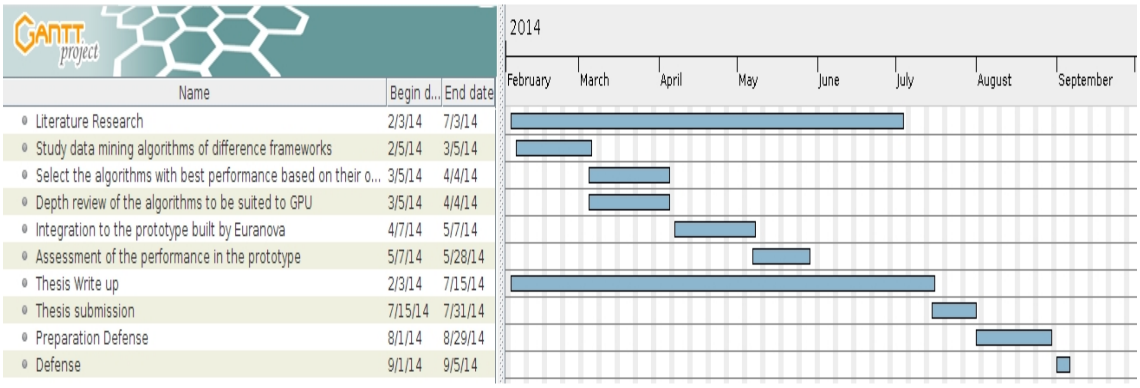

Figure 1.4: Initial Planning for the master thesis

1.6 Thesis Structure

In the following chapters, relevant background information will be provided. Chapter 2 provides scientific background for GPU programming as well as some good practices using Thrust library. In chapter 3, a brief introduction of Map Reduce is presented since it was the parallel technique used in the K-Means CPU implementation of Mahout. In chapter 4, information regarding Data Mining is presented, containing K-Means and Expectation Maximization, as well as the parallel implementation of K-Means using the SNIFF model. Related Work is also presented in this chapter, which includes an analysis of different frameworks making K-Means available and Literature survey. Chapter 5 contains the contributions of the thesis. This chapter includes a review of the requirements for the best practices using GPU and Data Mining, an analysis for the data mining frameworks regarding K-Means, as well as the methodology that was followed for optimizing SNIFF K-Means, further optimizations and the Expectation Maximization in the SNIFF model. In chapter 6, Final planning will mention some minor changes of the initial planning. Finally, chapter 7 includes Conclusions & Future Work.

Chapter 2

GPU Programming

In this chapter, we will explain basic concepts of GPU Programming. We will highlight some important structures illustrated with examples, which are very common at programming with GPU using Thrust Library.

2.1 GPU Programming

At first, GPUs were only used as 3D accelerators; nowadays, GPUs are also being used to speed up general-purpose computations. Due to their massive parallelism, they are well-suited for executing many similar tasks in parallel [1]. In this section, characteristics of the GPU are presented as well as the GPU programming model. We will mention some of the main advantages and disadvantages of programming with GPU. An extension of C programming language called CUDA will be presented since it was the one used during the experiments of SNIFF due to its simplicity and performance compared to other languages using GPU [39] such as OpenCL. Finally, Thrust Library, which simplifies CUDA programming, will be presented.

2.1.1 GPU

GPU [16] is a processor which is in charge of handling graphical tasks. A GPU is similar to a CPU. Nevertheless, a GPU is specifically designed to perform complex mathematical and geometric calculations that are required for graphics rendering, since it can draw multiple pixels in parallel. CPU and GPU process their tasks differently. A CPU contains few cores optimized for sequential serial processing while a GPU consists of thousands of smaller cores designed for handling multiple tasks simultaneously [40]. At the beginning, GPUs were made specially for graphics accelerators; nevertheless, their power can also be used for non-graphics applications, called General-Purpose Computing on Graphics Processing Units [1].

2.1.2 GPGPU - General-Purpose computing on Graphics Processing Units

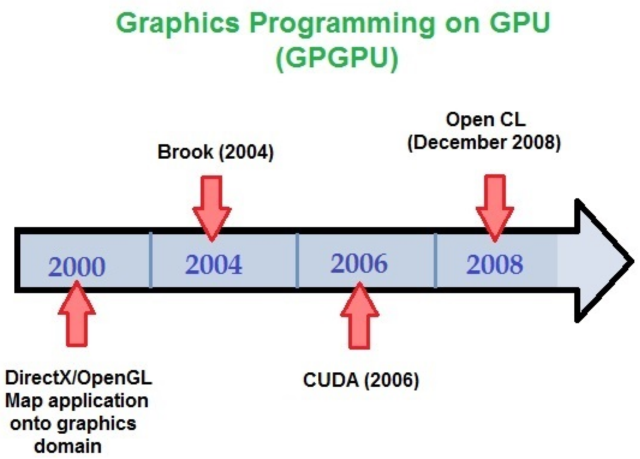

FIGURE 2.1: GPGPU Timeline adapted from [4].

At first, GPUs were used as graphics accelerators; however, in 2000, researchers started to use these graphics adapters for non-graphics applications [25]. Programming for the GPU [25] was very challenging because deep knowledge of the GPU architecture was required and it was available only for people who were familiar with graphics processing language, e.g. OpenGL or Direct3D. However, Ian Buck et al. [41] developed a system for General Purpose Computation on GPU called Brook. Brook was a C extension that made easier for a wider audience to use GPU Processing. Ian Buck and other developers released CUDA in 2006. CUDA is a parallel computing platform and programming model for general-purpose computation. In 2008, the Khronos Group released a cross-platform parallel programming model OpenCL 1.0 a competitor of CUDA.

2.1.3 GPU Programming model

A program for GPU is separated into two parts [25]: The host code and the kernels. The host code is executed on the CPU and the kernels on the GPU. The host code in the CPU will be in charge of running the main program and sending directions to the GPU, launching the kernels. A kernel is a program that is executed on the GPU; all threads in a kernel perform the same tasks, the difference is the data that is processed. A kernel is a grid of blocks, where each block is assigned to a streaming multiprocessor (SM) and can contain 1024 threads as maximum. The execution is divided into groups of 32 threads, called warps, that run at the same time, instead of executing the 1024 threads that are on a block [25]. The number of warps composed of 32 threads can be many in the GPU depending on the number of cores of the GPU. These will be executed in parallel and the remaining threads will be waiting till the others warps finish in order to be executed. The following figure represents a kernel launch that will store a block of threads in a grid and in each block containing these threads [5]:

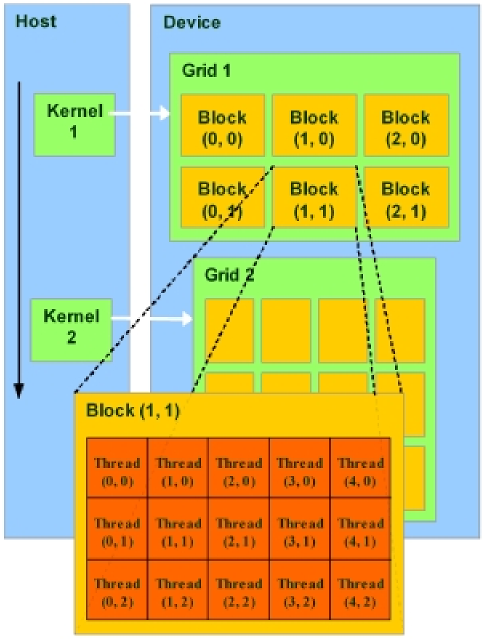

FIGURE 2.2: GPU kernel structure and invocation [5].

In order to have a great performance when running the kernel, these are some of the challenges proposed by [25]:

1. Graphics devices only offer limited synchronization possibilities.
2. Memory access should be coalesced to fully utilize the high bandwidth of device memory. Coalesced is an access pattern where each thread access a consecutive block of memory without any gaps. This results in a great performance since whenever each thread wants to read or write to the global memory, it always access a large chunk of memory; the GPU can reuse this large chunk of memory for all the threads that are contiguous, giving better performance than if the access was randomly done [6]. As can be seen in the following pictures:

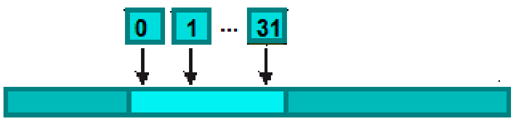

FIGURE 2.3: The threads are contiguous having good performance [6].

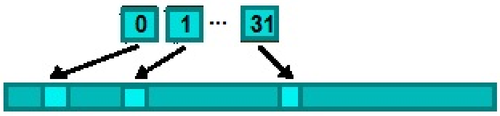

FIGURE 2.4: The threads are random having bad performance [6].

3. Divergence at warp level should be avoided.

Divergence [25] in the GPU is present when some threads take a different execution path when evaluating a condition. When threads take different code paths, since there is a single program counter per SM, the execution is serialized and part of the processor idle. The best way to have a good performance is that all threads that are assigned to a SM execute the same instruction per cycle. In the case of inter-warp, since they are in different warps, divergence is not an issue. This execution in GPU is called SIMT (Single Instruction, Multiple Threads). The CPU programming model [25] follows the execution of SIMD (Single Instruction, Multiple Data), in which it allows each thread to branch.

2.1.4 Advantages and Disadvantages of programming with GPU

Some advantages and disadvantages of programming with GPU are presented in the following table [25, 42]:

Table 2.1 GPU Advantages and Disadvantages

Advantages	Disadvantages
1. GPUs can run algorithms from 10 to 100 or more times faster than a CPUs. 2. GPU technology is now available for reasonable prices.	1. Programming for GPU differs significantly from traditional CPU. 2. Incorporating GPU hardware into systems adds expense in terms of power consumption, heat production and cost. 3. Data transfer at the bus between CPU and GPU may have performance problems due to the Peripheral Component Interconnect Express (PCIe) bottleneck.

2.1.5 CUDA

CUDA [18] is a parallel computing [43] platform and programming model created by Nvidia; it improves the computing performance by harnessing the power of the GPU. CUDA takes advantage of heterogeneous systems that have both CPUs and GPUs [44]. A CUDA program is split into two parts: The host code, which is the code executed in the traditional CPU, and The device code, which runs on the GPU [44].

We illustrate the advantage of using GPU vs CPU with an example from [45]. In this example, 100 numbers need to be squared. The following fragment shows how to code this task for CPU:

```
for(i=0; i<100; i++)
{
        output[i] = input[i] * input[i];
}
```

This previous code is only launching one thread at a time in which is putting the result of the multiplication done by the input array in the output array, and it loops 100x times, one thread after another. In the case of the GPU, it is possible to just run a program called kernel in which it will launch for this example 100 threads at the same time in a block, as oppose to the previous CPU example, making the square of each number of the array, as shown with the following code:

```
square<<<1, 100>>>(d_out, d_in); //Kernel, launching 100 threads in 1 block
square(output, input){
idx = threadIdx.x;                //Each thread contains an id.
f = input[idx];                  //Based on this id, the value is stored in f
output[idx] = f * f;              //The square of the numbers in
                                 //f is stored in output array
}
```

Appendix B contains the full implementation of the CUDA code retrieved from [45]. It describes all the previous procedures for launching the kernel, such as memory allocation from the CPU to the GPU. Note that to launch a Kernel you have to allocate memory from CPU to the GPU and then process the data in the GPU. Finally, the result must be copied back from the GPU to the CPU. In CUDA, there is a library called Thrust [46], which allows the users to implement parallel applications, making it easier to be implemented since it reduces many complexities that will be explained in the following section.

2.1.6 Thrust

Thrust is a library based on the C++ standard template library and it is fully interoperable with CUDA [46]. Thrust helps the programmers to write GPU applications efficiently and with minimum effort, since it contains very efficient implementations of algorithms such as sort, transform, reduce, etc.

Using Thrust, the developers describe their computation using the algorithms that are provided by the library and completely delegate how to implement the computation to Thrust. This allows the programmers to describe only what to compute without additional restrictions on how to carry out the computation [20]. For example by not having to deal with all the detailed complexity of memory allocation such as in pure CUDA code.

Features of Thrust can be presented in the following examples that were retrieved from [20]. A Thrust program which sort data on the GPU [20]:

```
int main (void)
{
        //generate 16M random numbers on the host
        thrust::host_vector <int> h_vec (1 << 24);
        // transfer data to the device
        thrust::device_vector <int> d_vec = h_vec;
        // sort data on the device
        thrust::sort(d_vec.begin(), d_vec.end());
        //transfer data back to host
        trust::copy(d_vec.begin(), d_vec.end(), h_vec.begin())
}
```

Thrust provides two vector containers: host_vector and device_vector. The host_vector is stored in the RAM memory and the device_vector in the memory of the GPU. As shown in the previous example, after generating random numbers in the host_vector it will allocate and transfer the data into the device memory, then the sorting function will be executed in the GPU for all the values contained in the device_vector. Finally, the result is transferred to the host_vector using the copy instruction from Thrust. Note that running the sort function may imply launching one or more CUDA Kernels but all these are abstracted, so the programmer does not have to deal with the launch configuration of the kernels, Thrust library does it for you [20].

Some of the common implementations and examples of Thrust parallel algorithms executed in the GPU along with their descriptions are presented next [46]:

2.1.6.1 Reduce

A reduction algorithm uses a binary operation to reduce an input sequence to a single value, e.g. the sum of an array of numbers is obtained by reducing the array. If no binary operation is specified, the elements will be summed by default (+ operation). As an example, suppose we have one input vector with these values:

Vector	10	20	50	20

The output of the following function:

int Sum = thrust::reduce(Vector.begin(), Vector.end());

will have a value with the sum of all the elements:

Sum	100

2.1.6.2 Sort

Thrust offers several functions to sort data or rearrange data according to a given criterion. By way of illustration, we have this input vector:

Vector	3	2	1	5	4

The output of the following function:

thrust::sort(Vector.begin(), Vector.end());

will be a sort the vector in ascending order:

Vector	1	2	3	4	5

2.1.6.3 Fill

Simply sets a range of elements to a specific value. For example, this input vector with 0 values:

Vector	0	0	0	0	0	0	0

After the following function:

thrust::fill(Vector.begin(), Vector.end(), 7);

The vector will contain the value 7 in all its elements:

Vector	7	7	7	7	7	7	7

2.1.6.4 Sequence

A function can be used to create a sequence of equally spaced values. As a case in point, this input vector with 0 values:

Vector	0	0	0	0	0	0	0

The following function:

thrust::sequence(Vector.begin(), Vector.end());

will fill the vector with the following sequence:

Vector	0	1	2	3	4	5	6

2.1.6.5 Transform

Transform applies a unary function to each element of an input sequence and stores the result in the corresponding position in an output sequence. Specifically, an operation is performed for each element i in the range and the result is assigned to the output iterator. The input and output sequences may coincide, resulting in an in-place transformation [47]. For instance, we have two vectors as input and one as output:

Vector	5	6	9	10	20	30
Vector2	2	3	5	5	10	20
Vector3	0	0	0	0	0	0

thrust::transform(Vector.begin(), Vector.end(), Vector2.begin(), Vector3.begin(), thrust::minus<float>());

This function will compare each element in the Vector and Vector2, and compute the difference and storing the result in Vector3. Having the following output for Vector3:

Vector3	3	3	4	5	10	10

Thrust covers most of the arithmetic operations (+, - , /, *, mod). However, often we want to do something different. For instance, there is a very well known vector operation called SAXPY operation: y <- a * x + y, where x and y are vectors and a is a constant. In this case we can implement our own function SAXPY to do this operations in a single transformation.

For example, we have as input the following two vectors and a constant "a":

Vector X	2	4	6	8	10
Vector Y	5	10	15	20	25
a	10				

Where saxpy is the function that will perform for a multiplication with a constant ”a” with x and a sum with Y. It is possible to compute more operations with each value in a vector just by following this saxpy example in thrust.

This function will perform the SAXPY operation y <- a * x + y :

thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy(a));

It will output the following vector storing the results in the four parameter (Y.begin()) of the transform operation:

Vector Y	25	50	75	100	125

All the values computed by the operation are stored in Y vector by overwriting the previous values in Y; preventing to allocate memory in another vector.

2.1.6.6 Sort_by_key

Sort_by_key performs a key-value sort. By way of illustration, these two input vectors:

Keys	1	3	2	6	4	5
Values	10	20	30	50	15	25

With the next function:

thrust::sort_by_key(Keys.begin(), Keys.end(), Values.begin());

The output of the keys and values is:

Keys	1	2	3	4	5	6
Values	10	30	20	15	25	50

2.1.6.7 Reduce_by_key

Reduce_by_key is a generalization of reduce to key-value pairs. For each group of consecutive keys in the range keys_first, keys_last that are equal, reduce_by_key copies the first element of the group to the keys_output. The corresponding values in the range are reduced using the plus operation and the result copied to values_output. For instance, these two input vector:

KeysInput	1	3	3	3	2	2	1
ValuesInput	9	8	7	6	5	4	3

With the following function:

thrust::reduce_by_key(KeysInput.begin(), KeysInput.end(), ValuesInput.begin(), KeysOutput.begin(), ValuesOutput.begin());

will put the result by grouping by same keys in the two output vectors: KeysOutput and ValuesOutput:

KeysOutput	1	3	2	1
ValuesOutput	9	21	9	3

2.1.6.8 Count

Count returns the number of instances of a specific value in a given sequence. For example, this input vector:

Vector	1	3	5	2	5	7	5

This function:

int Result = thrust::count(Vector.begin(), Vector.end(), 5);

will assign to the result variable, the total number of elements with value 5:

Result	3

2.1.6.9 Repeated_Range

Repeated_Range will create a vector with the number of repeated values according to a parameter as output. For example, the next input vector:

Vector	1	2	3	4

By assigning the value three for repetition with the following function:

repeated_range <Iterator>Vector2(Vector.begin(), Vector.end(), 3);

The output will be a vector with repeated range of three for each element.

Vector2	1	1	1	2	2	2	3	3	3	4	4	4

Chapter 3

MapReduce

In this chapter, a brief introduction of Map Reduce is presented. It is the framework used by Mahout to process the data, in which the Mahout CPU implementation of K-Means was compared to the SNIFF GPU K-Means implementation.

3.1 MapReduce

Distributed systems can present many complexities [48], such as computation parallelization, work distribution, and dealing with unreliable hardware and software. Nevertheless, the MapReduce model makes parallel processing simpler by abstracting away from these complexities. MapReduce uses key/value pairs: The keys are used to identify the information and the values are the data associated with the key.

MapReduce [49] is a programming model for processing large datasets. MapReduce allows to write data-parallel programs and execute them in a distributed system.

In MapReduce a user implements two main phases [7, 50]:

1. Map Phase. - The node takes the input in a set of key/value pairs, the map will produce zero or more key/value pairs as output to the Reduce Phase.
2. Reduce Phase. - The node with the reduce phase will gather all the output passed through the map phase and combine them in order to form the global output.

Between the Map Phase and the Reduce phase, there is a shuffle phase taking place. This sorts the resulting key/value pairs from the map phase by their keys; then, these pairs will be assigned to the reducer according to their keys [7].

A MapReduce program breaks down the work submitted by a client into a small parallelized map and reduces tasks, also called mappers and reducers. Each map task deal with a relatively small amount of data and work in parallel. The output of the map tasks are the intermediate key/value pairs that can be group by same keys before being sent to the reduce phase [7]. This grouping is also called combiner, which is an optimization by reducing the transfer time in the shuffle phase to the reduce tasks, which represent synchronization of the map tasks [7]. The following figure from [7] shows an iteration of the three phases previously explained:

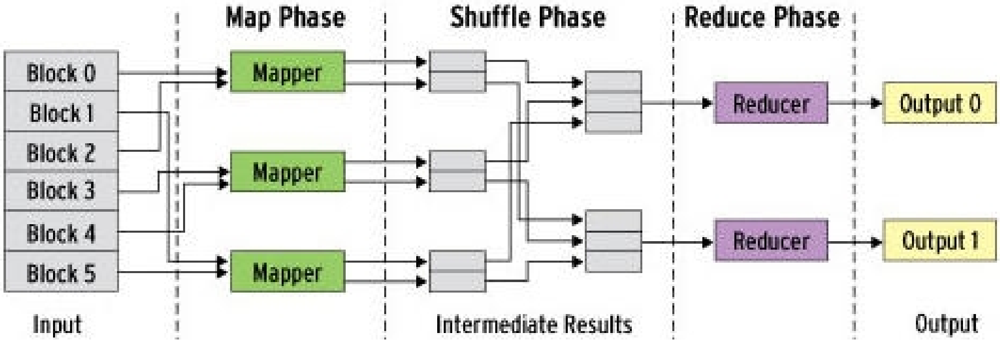

FIGURE 3.1: The MapReduce framework breaks down data processing into map, shuffle, and reduce phases [7].

Word count is a typical example used for understanding MapReduce. The following figure from [8] explains the flow of using MapReduce in word count when having as input a file with 4 lines of text with a set of 3 words each:

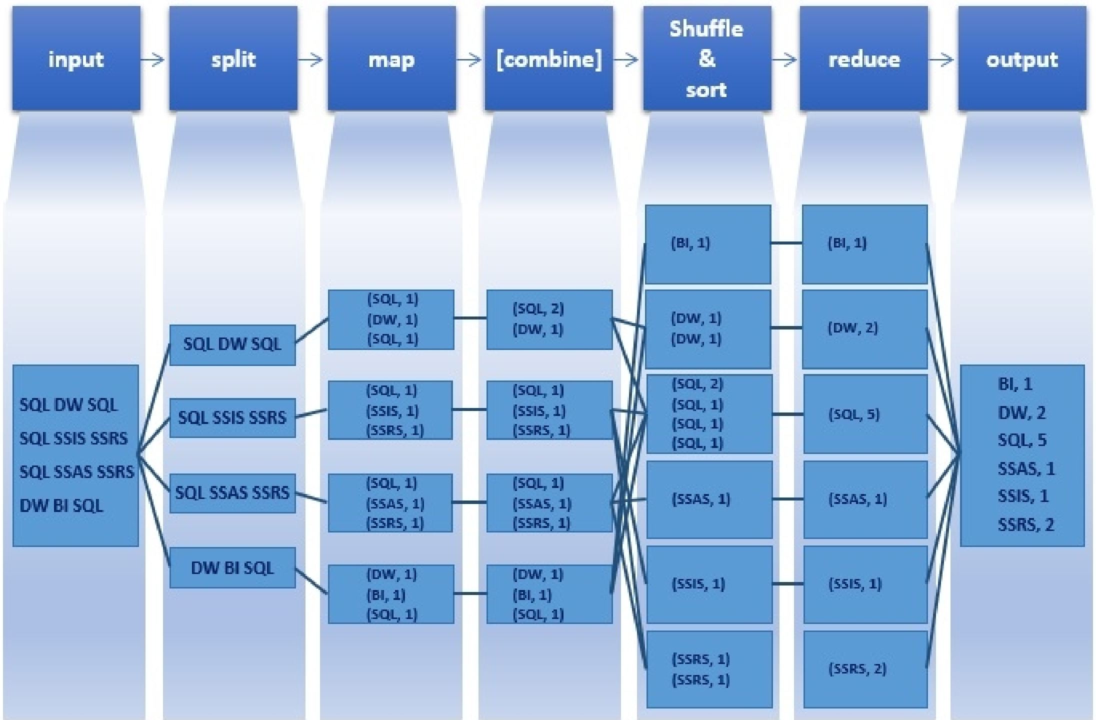

FIGURE 3.2: MapReduce Word Count example flow [8].

The steps of the figure from [8] are the following:

1. **Input:** A file with 4 lines of text is used as input to MapReduce.
2. **Split:** For this example, the file is split into four sets, each set is one line from the input file.
3. **Map:** Each split is fed to a map() function implemented by the user, specifying the logic of how this input data will be processed. For this example, the map() function will count the occurrence of each word and adapt it to a (key, value) pair.
4. **Combine:** Optional step and is often used as an optimization to reduce the data transferred across the network. It is accumulating the values per mapper instead of sending each result of the map to the shuffle, thus reducing the transfer time.
5. **Shuffle & Sort:** Output of all the map() function is collected, shuffled, and sorted and ready to be sent to the reducer phase.
6. **Reduce:** The data from many map() function is aggregated and the word counts are built as (key, value) pairs.
7. **Output:** The output of the reduce phase is written to a file.

The main contributions of the MapReduce programming models are the scalability and fault-tolerance achieved for a variety of applications that fit this model [49].

Chapter 4

DATA MINING

In this chapter, background for the data mining, K-Means and Expectation Maximization algorithms will be further explained, as well as information regarding the data mining frameworks such as Mahout, R and Weka. We will also present how the implementation of K-Means using GPU in SNIFF proposed by Tran et al [1], related work and literature survey.

4.1 Data Mining

Data mining is the process that can be used to discover useful and interesting patterns and relationships in huge quantities of data to generate rich knowledge and help organisations, government and individuals [51].

The Knowledge Discovery Process [52] is a broad process for finding knowledge in data, as illustrated in the following figure [51]:

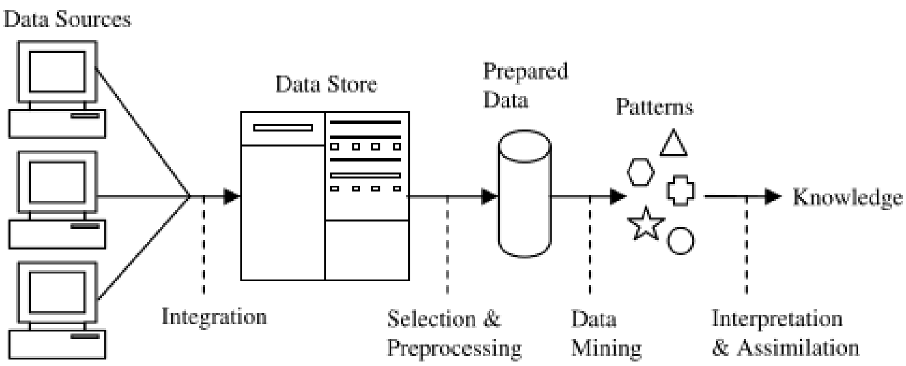

FIGURE 4.1: The Knowledge Discovery Process

The Knowledge Discovery Process starts with the integration of the data, which can come from many sources. This integration places the data into a data store such as databases or in flat files. Next, some part of the data will be selected and pre-processed into a standard format. The data mining algorithms will discover some patterns and produce an output, which will be interpreted in order to generate relevant knowledge [51]. Data Mining is just one part of the knowledge Discovery process.

Data Mining methods may be classified as either supervised or unsupervised [53]. In supervised methods, there is a particular target variable and the algorithm will have many examples where the value of the target variable is contained, meaning that the algorithm can learn from previous data. In the case of the unsupervised methods there is no target variable. Rather, the data mining algorithm searches for patterns in all the variables. The most common unsupervised data mining algorithm is clustering [53]. Clustering is the task of grouping objects which are similar to each other [51].

In the following section, two unsupervised learning algorithms will be explained: the K-Means algorithm [23] and Expectation Maximization [38].

4.1.1 K-Means

K-means clustering is an unsupervised learning algorithm [51]. The objects have to be assigned to just one of a set of clusters (group of objects). For this clustering algorithm [51], first, it is necessary to indicate the number of clusters that we want to create from the data. This number of clusters is called K. Most of the time, the value of K has a small integer value, e.g. 2,3,4, but it can be larger. It is possible to measure the quality of the clusters formed by taking the sum of the squares of the distances of each point from the centroid of the cluster to which is assigned and computing the average. We can randomly select K-clusters, but the method may work better if we pick K initial points that are fairly far apart. After selecting K initial points as shown in the following figure:

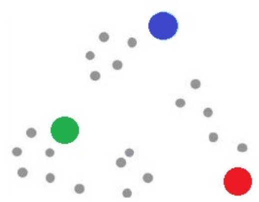

FIGURE 4.2: Initial centroids (Green, Blue and Red dots) and datapoints (gray dots) [9].

We assign each of the points to the cluster which has the nearest centroid. All the objects have to be assigned to just one cluster:

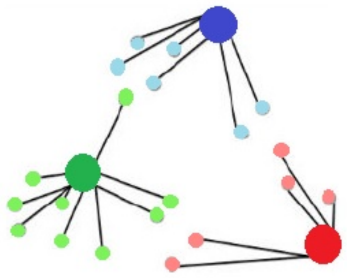

FIGURE 4.3: Each datapoint is assigned to its nearest centroid [9].

Then, based on the assignation of the points to the clusters, we recalculate the centroids of the clusters:

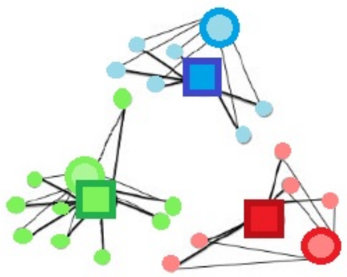

FIGURE 4.4: Recalculate the centroids based on the datapoints assignations [9].

Subsequently, we get the new cluster values:

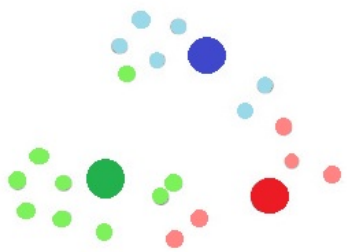

FIGURE 4.5: New cluster values [9].

We keep repeating these steps until it reaches a max-limit iteration criterion or when the algorithm converges. The K-Means algorithm can do multiple rounds of processing and produces new centroids locations after each iteration [9].

In the K-Means algorithm, it is not possible to determine which number of K will produce the best set of clusters [51]. The initial selection of centroids can affect the result. This is the reason why we should run several times the algorithm for a given value of K, each time with a different choice of the initial K centroids [51].

4.1.2 Expectation Maximization

The Expectation Maximization (EM) [54] is an unsupervised clustering method which follows an iterative approach, which attempts to find the parameters of the probability distribution that has the maximum likelihood of its attributes.

Maximum-likelihood [54] can provide an estimation for the model's parameters, such as the mean and variance, by applying this method in an observed data set and a statistical model; e.g. the Mixture model. Mixture models [10] are basically a probabilistically way of doing soft clustering, in which the datapoints can belong to any of the clusters. For each cluster, it corresponds a probability distribution, and with the EM algorithm you can discover all the parameters of the distribution for each cluster.

The following figures from [10] explain the basic concepts of Expectation Maximization.

For example, there are some datapoints in which is already known that come from two gaussian models and also from which model they belong to. Thus, estimating the mean and the standard deviation for each Gaussian is trivial.

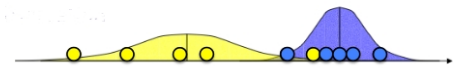

FIGURE 4.6: Observations in which the gaussian models are already known [10].

In the following example, it is not known the gaussian where each datapoint belongs to. Then, EM can be used to infere from which gaussian each datapoint more likely belongs to, and compute the mean and the standard deviation.

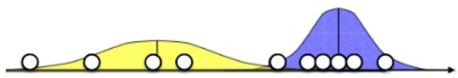

FIGURE 4.7: Observations where the gaussians are not known [10].

In this example, suppose we have some datapoints and two gaussian models that were put randomly in space; much like the K-Means algorithm with the K clusters.

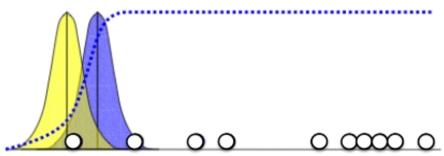

FIGURE 4.8: First iteration [10].

In the EM applying soft clustering, we estimate the probability from where each datapoint more likely belongs to a certain distribution. In this case, the first datapoint is represented in full yellow color, since it is more likely that it belongs to the yellow gaussian for being closer to its mean. The next four datapoints have a certain probability that belong to either yellow or blue gaussian according to their mean. Lastly, the following five datapoints are in full blue color since they are more likely to belong to the blue gaussian. Note: The points cannot have a 100% probability of belonging to one distribution, but in the example from [10] is shown that the closer one datapoint is to the mean of some gaussian, the more likely this datapoint belongs to that gaussian.

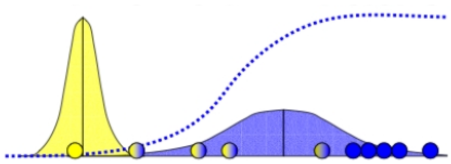

FIGURE 4.9: Result of first iteration [10]

This figure shows the result of the new gaussian or means of the second iteration, estimating again the probabilities using EM.

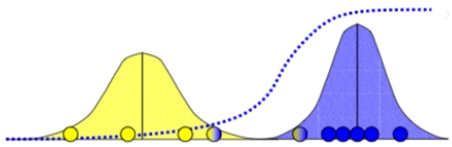

FIGURE 4.10: Result of second iteration [10]

EM is an extension from the basic approach of clustering [11] because, rather than assigning observations to clusters to maximize the differences in means, EM algorithm computes probabilities for each cluster based on one or more probability distributions. The objective of the EM clustering algorithm is to maximize the likelihood of the data, given the clusters [11]. In summary, EM [11] begins with an initial estimate for the missing variables and keeps iterating. In the expectation step, it finds the maximum likelihood for these variables and in the maximization step, it computes the means or the new clusters.

Expectation maximization for soft-clustering can be implemented containing four major steps [11]. To initialize, we guess the means and standard deviations; the means represent the clusters. We start with a ML hypothesis through an iterative scheme to find the final clusters k.

1. Initialization step: Initialize the hypothesis is $\theta^0 = (\ \mu_1^0, \mu_2^0, \ldots\ , \mu_k^0\ ) \rightarrow \Theta_k^0 = \mu_k^0$ where: K is the number of Gaussians. σ is the standard deviation, θ^0 is the estimate at 0^{th} iteration, μ is the mean.

2. Expectation step: Estimate the probabilities Z_{ik} of the hidden variables with respect to the known means and standard deviations from the initialization step.

Z_{ik} will contain the probabilities of an unobserved variable i, which belongs to cluster k.

$$E(z_{ik}) = \frac{\exp\left[-\frac{(x_i - \mu_k^t)^2}{2\sigma^2}\right]}{\sum_{j=1}^{K} \exp\left[-\frac{(x_i - \mu_j^t)^2}{2\sigma^2}\right]}$$

FIGURE 4.11: Expectation step [11].

where: t represents the number of iteration, the values of $E(Z_{ik})$ are the probabilities for the hidden variables (mean and standard deviation), k is the number of the cluster, σ is the standard deviation.

3. Maximization step: Compute the new estimate of the parameters which will give the new clusters.

$$\mu_k^{t+1} = \frac{\sum_{i=1}^{n} E(z_{ik}) x_i}{\sum_{i=1}^{n} E(z_{ik})}$$

FIGURE 4.12: Maximization step [11].

4. Convergence step: if $||\theta^{t+1} - \theta^t|| <$ e. check whether it converged based on a minimum threshold or if it reached a maximum number of iterations; otherwise, keep repeating the expectation and maximization step.

4.1.3 Data Mining Frameworks

R [35] is a free programming language and software environment for statistical computing and graphics. Statisticians and data miners are widely using this language for data analysis. R contains numerous statistical techniques, e.g. linear and nonlinear modeling, classical statistical tests, classification, clustering, etc. One of the main strengths of R is the facility for producing high quality plots. R is easily extensible through functions and extensions. For example, advanced users can write C, C++, Java, or Python code to manipulate R objects directly [35].

Through the uses of packages, R supports a lot of data mining techniques [55], which includes: Classification and prediction, regression, clustering, text mining, Time Series Analysis, etc. Yanchang Zhao [56] presents a guide to R for data mining in a Reference Sheet, which provides a comprehensive index of R packages and functions for data mining, categorized by their functionality.

Weka [36] is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from your own Java code. Weka contains machine learning algorithms such as classification, regression, clustering, association rules, etc [36]. These algorithms can be applied through Java code or, by the use of a graphic user interface, can be applied directly to the datasets.

Mahout [57] is a scalable machine learning library which is widely used to process large data sets. Mahout contains algorithms such as clustering, classification and collaborative filtering which can be executed in distributed systems. They also have a version of these algorithms that can run on a single machine as well.

At the moment, Mahout supports mainly three use cases [57]:

1. Recommendation mining. - It suggests items to the users that they may like, based on understanding the user's behaviour.

2. Clustering. - Takes, for example, text documents and groups them based on their topic.

3. Classification. - It learns from previously categorized documents which documents, of a specific category, look alike and for those documents that are unlabelled, it tries to assign them a label.

4.2 SNIFF - Parallel Implementation of K-Means using GPU

In the following section, the parallel implementation of K-Means using GPU by SNIFF will be explained. First, SNIFF architecture will be presented in order to understand how the K-Means algorithm was implemented based on the framework.

SNIFF framework can run data processing algorithms formed by the following phases [1]:

1. Independent computation of a partial model in separate data chunks.
2. Combination of the partial models in order to compose the global model.
3. Iterate again if needed, e.g. K-Means, until a convergence criteria is met.

The following figure shows SNIFF architecture [1]:

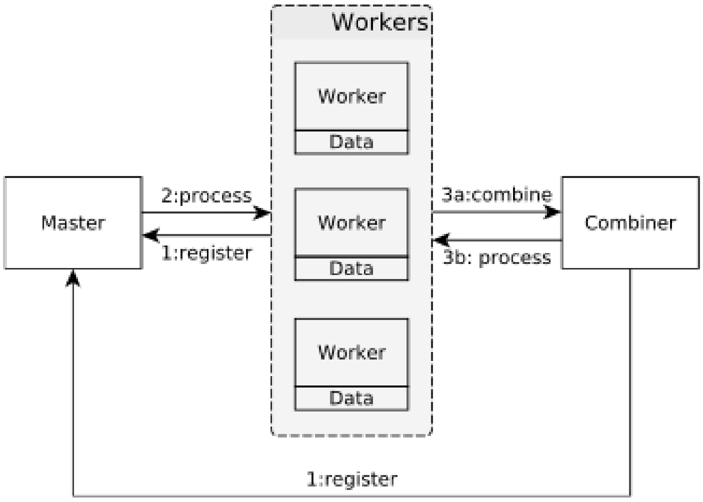

FIGURE 4.13: Architecture of SNIFF framework. All workers are equipped with a GPU and in order to process the data, each worker needs to register to a master node, in which they receive instructions, job information and cluster configuration. Step 3 a and b may be repeated [1].

Tran et al. [1] implemented a parallel K-Means algorithm based on the distributed version by [58] in the SNIFF architecture. In the SNIFF implementation, the whole dataset is partitioned equally among all the workers. Then, each node receives the same initial centroids and performs the k-means iterations on the partial data. When the iteration ends, the updated centroids position and a counter containing the data points belonging to each cluster are collected by the combiner; finally, the combiner will globally compute the centroids positions.

All the operations in K-Means algorithm are data-intensive [59] operations which can be parallelized on the GPU device. Each iteration performs the following steps in the worker node [1]:

1. Compute the distance of each data point to each of the K centroids.
2. Find the closest centroid for each data point, and use counters containing the number of data points belonging to each cluster.
3. Update each centroid according to the data points coordinates belonging to each cluster.

All the previous steps were designed in the SNIFF K-Means implementation to run on the GPU. The following figures represent the routines for the SNIFF distributed K-Means in the Worker and in the Combiner [1]. For the worker routine, we used the variables x_i to represent the datapoint and m_j for the clusters.

Algorithm 1 Distributed k-Means - Worker Routine

```
Read data T from local data partition in GPU memory
Set active = true
while active do
    centroids ←Receive k centroids {m_j} from combiner
    for all centroids m_j do
        D_j ← Compute all d^2(x_i, m_j)
    end for
    idx ← Identify index j of centroid closest to x_i
    ct'_j ← Increment count number of matched centroid
    MSE' ← Sum all d^2(x_i, m_j) for x_i to its closest m_j
    for all m_j do
        {m'_j} ← Sum each x_i coordinates to its matching m_j
    end for
    Send {m'_j}, ct'_j and MSE' to Combiner
    active ← get active response from Combiner
end while
```

FIGURE 4.14: Distributed K-Means - Worker Routine [1].

Algorithm 2 Distributed k-Means - Combiner Routine

```
Generate k initial centroids {m_j}
MSE ← Large number
while MSE > threshold do
    Send {m_j} and active to the w workers
    Receive model parts {m_j}', ct'_j and MSE' from all w workers
    ct_j ← Σ ct'_j
    m_j ← (Σ m'_j) / ct_j
    MSE ← Σ MSE'
end while
Send active ← false to the w workers
```

FIGURE 4.15: Distributed K-Means - Combiner Routine [1].

4.3 Related Work

Lots of works have been going on under K-Means in different frameworks. We will also present a literature survey of K-Means GPU implementations.

4.3.1 K-Means Frameworks Implementations

In this section, we will highlight Mahout parallel K-Means implementation. Sequential K-Means implementations of R and Weka will also be discussed.

4.3.1.1 Mahout Parallel K-Means

Today, there is a need to cluster large datasets that cannot fit into memory. Mahout's K-Means implementation can be used for large datasets using the Hadoop framework [12].

In order for Mahout to process the data, it is necessary to have it in numerical representation, if the data is not numerical, it will need to be preprocessed. Afterwards, this data should be converted into vectors and finally into a sequenceFile which is a specific Hadoop file format [12].

The K-means implementation in Mahout receives the following input parameters [9]:

- A SequenceFile containing the input vectors.
- A SequenceFile with the initial cluster centers, which can be random.
- A similarity measure, e.g. Euclidean distance, Cosine distance, Manhattan distance, etc.
- A convergence threshold.
- A maximum number of iterations in case there is no convergence first.
- The vector implementation used for the input files.

As an output, you get the final centroids coordinates and the samples attributed to each cluster. The output files are in SequenceFile format. Mahout provides some libraries in order to convert txt files and create the vectors that will be the input for the K-Means implementation [12].

According to [12], Mahout K-means contains three stages:

1. Initial stage: the dataset will be split into many blocks.
2. Map stage: In each map function, it reads the centroids in memory first, and then it will calculate the distances between the objects and the cluster centroids; the objects will be assigned to the nearest cluster centroid. Each map task is processed within a data block. Each key represents a datapoint where it will store the index to which cluster it belongs to, and the value contains the dimensions values of the datapoint.
3. Reduce stage: In the reduce function, the cluster's centroid will be recalculated in that cluster. Also for each of the clusters in the reduce phase it will calculate whether the cluster converged or not. If the algorithm converged, then the iterations loop finish, otherwise it will continue until it reaches the maximum number of iterations.

The three stages of Mahout K-means are represented in the following figure from [12] :

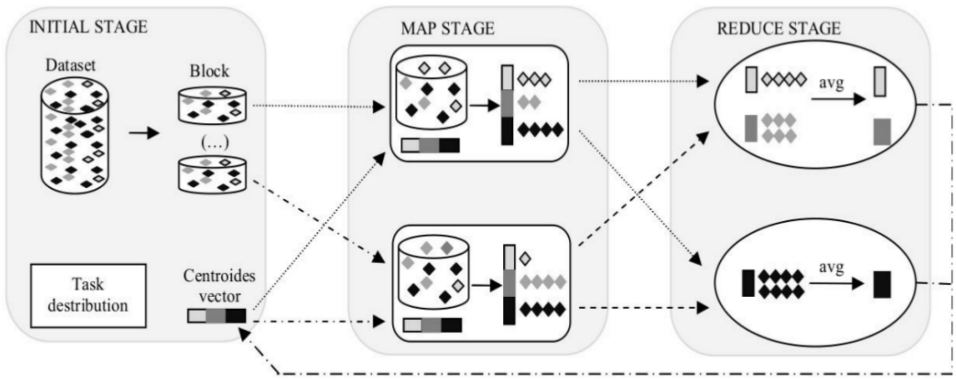

FIGURE 4.16: Mahout K-Means Implementation [12].

In Mahout there are two types of implementation of K-means: Sequential and using MapReduce. The focus will be in the Map-Reduce implementation, since it is distributed in different nodes, like the SNIFF K-Means.

The main steps of the pseudo code were taken from [60] and from the implementation in version 9 of Mahout K-Means [24] which are located in lines [4,7-8,11-18]. Each instance represents a record in the dataset. The following steps describe the Mahout K-Means implementation at a higher level, whereas an instance represents just one datapoint:

Algorithm 3 Map Phase (key, value)

```
1: Input: Global centroids and the sample value
2: Output: A (Key', value') pair, where key' is the index of the closest center point
   and value' is a string containing all the dimensions values of the datapoint.
3: Construct the sample instance from value;
4: maxProb = Double.Min_VALUE;
5: Index = −1;
6: for all i=0 to Centers.length do
7:     dis ← sqrt.(d(instance, centers[i]));
8:     pdf[i] ← 1/(1 + dis);
9:     totalPdf ← totalPdf + pdf[i];
10: end for
11: Norm = 1/totalPdf;
12: for all i=0 to centers.length do
13:     prob[i] ← pdf[i] ∗ Norm;
14:     if prob[i] > maxProb then
15:         maxProb ← prob[i];
16:         index ← i;
17:     end if
18: end for
19: Take index as key'
20: Construct value' as string comprise of the value of different dimensions; output
    (Key', Value')
```

FIGURE 4.17: Distributed K-Means - Map Phase.

Note that in line 7, an square root is applied after the distance computation of the datapoint with the centers. Also, instead of using the minimum distance to compute the new centroid, lines [8-9, 11-13] are transforming the distance into a probability, where the centroid closest to the instance will contain higher probability. On the contrary, the instance that are further from the centroids will have less probability. Lines [14-16] will assign to the index variable the position of the cluster for the instance which contains the highest probability.

Mahout's K-Means implementation also contains a combiner which is an optimization that will partially sum the instances assigned to the same cluster; thus, decreasing the number of intermediate keys that are being passed to the reducer. To compute the global mean, it is necessary to record the number of samples in the same cluster of the same map task. Figure 4.18 is the pseudo code of the combiner from [60]. Lines [5-8] are accumulating the instances in an array with the same cluster index.

```
Algorithm 4 Combine (key, V)
Input: key is the index of the cluster, V is the list of the samples assigned to the same cluster.
Output: ( key', value' ) pair, where the key' is the index of the cluster, value' is a string containing a sum of the samples in the same cluster and a count of the samples.
An array containing the samples in the list V;
A counter num, which will keep record of the sum of sample number in the same cluster;
while V.hasNext() do
    Construct the sample instance from V.next();
    Add the values of different dimensions of instance to the array.
    num++;
end while
key' is the index of the cluster represented by key;
Construct value' as a string with the sum of the values containing different dimensions and num;
output (key', value') pair;
```

FIGURE 4.18: Distributed K-Means - Combiner Phase.

Finally, the input of the reduce function is the data obtained from the combine function of each node. As a result, the global centroids are computed [60] in figure 4.19. Lines [5-8] sum the values of all instances with the same cluster index from the output of the combiner. Line 10 is computing the new centroids.

```
Algorithm 5 Reduce Phase (key, V)
1: Input: Key is the index of the cluster. V is the list of the partial sums from different nodes.
2: Output: ( key', value' ) pair, where the key' is the index of the cluster, value' is a string representing the new centroid.
3: Initialize one array to record the sum of value of each dimension of the samples contained in the same cluster;
4: Initialize a counter NUM to record the sum of sample number;
5: while V.hasNext() do
6:     Construct the sample instance from V.next()
7:     Add the values of different dimensions of instance to the array
8:     NUM += num;
9: end while
10: Divide the entries of the array by NUM to get the new center's coordinates;
11: key' is the index of the cluster represented by key;
12: Construct value' as string comprise of the center's coordinates;
13: output (key', value') pair;
```

FIGURE 4.19: Distributed K-Means Mahout- Reduce Phase.

4.3.1.2 R K-Means

R provides many implementations of the K-Means algorithm, such as the Lloyd implementation which is the first and the simplest of all the clustering algorithms provided by R. Other implementations such as Forgy, MacQueen and Hartigan-Wong are also provided [35].

In R you can specify which of these implementations you want to perform. If no option is specified, then Hartigan and Wong will be executed, which is a variant of Lloyd algorithm [13]. The R implementation of the Lloyd algorithm was the one analyzed, since is the most similar to the one executed by EURA NOVA.

When performing R K-Means, there are some parameters that are needed [35]:

1. A numeric matrix x or an object that can be converted to such a matrix. This matrix x will be taken as our vectors to be clustered.
2. The number of K cluster centers or a set of initial cluster centers. If a number is specified, then it will randomly take some rows from the x matrix.
3. The maximum number of iterations allowed.
4. The algorithm to be used.

If there are no errors during the execution of the Lloyd K-Means, then it will return an object that will provide us with the following components [13]:

1. A numerical vector that will indicate for each point to which cluster it is allocated.
2. A matrix with the cluster centers.
3. The sum of squares errors for each cluster.
4. The number of points that are allocated in each cluster.

In the K-means implementations made by R, there is no convergence threshold, which means that it will only stop iterating if there is no relocation for the cluster or if the maximum number of iterations has been reached. The implementation of K-Means in R can display messages in case there are some exceptions during the algorithm execution, e. g. the number of clusters K is greater than the rows in x matrix.

Lloyd R K-means steps are based on the pseudocode in [13] and in the implementation of Lloyd R K-Means. This implementation stops iterating when a max number of iterations has been reached as shown in line 6, or if there was no update in any of the clusters for the given iteration as presented in line [22-23]. Lines [8-11] computes the distances for each of the clusters and datapoints. Lines [12-14] assigns the datapoint with the minimum distance to a cluster. Lines [25-27] update the clusters values.

Algorithm 6 R K-Means

```
1: N (number of datapoints)
2: MaxIter (limit of iterations)
3: for all i = 0; i < N; i++ do
4:     cluster[i] = −1
5: end for
6: for all iter = 0; iter < maxIter; iter++ do
7:     Updated = False
8:     for all i = 0; i < N; i++ do
9:         MinDist = Double.MaxValue
10:        for all j = 0; j < cluster.length; j++ do
11:            dist = computeDist(datapoint, centers[i])
12:            if dist < minDist then
13:                minDis = dist
14:                index = i
15:            end if
16:        end for
17:        if cluster[i]! = index then
18:            updated = true
19:            cluster[i] = index
20:        end if
21:    end for
22:    if !updated then
23:        Break;
24:    end if
25:    for all i = 0; i < cluster.length; i++ do
26:        UpdateCluster(cluster[i]);
27:    end for
28: end for
```

FIGURE 4.20: K-Means R implementation [13].

4.3.1.3 Weka K-Means

Some implementations of K-Means only accept numerical values for attributes. In the case of Weka, this one provides filters to accomplish all the preprocessing tasks. Weka K-Means implementation can handle a combination of categorical and numerical attributes. The algorithm normalizes numerical attributes when doing distance computations. Weka uses Euclidean distance to compute the distances between instances and clusters [61]. Weka K-Means is similar to R, in the sense that they both stop if no object moves and the centroids are not recalculating, instead of a threshold which is included in the implementation of Mahout. Also, the distance computation uses euclidean distance and takes the square root between coordinates and pairs of objects [61].

The implementation is based on the steps in [61] which is quite similar to the Lloyd R K-Means implementation [13], the most important change, in terms of computation, is that is doing the square root in the distance in line 11.

Algorithm 7 Weka K-Means

```
1: N (number of datapoints)
2: MaxIter (limit of iterations)
3: for all i = 0; i < N; i++ do
4:     cluster[i] = −1
5: end for
6: for all iter = 0; iter < maxIter; iter++ do
7:     Updated = False
8:     for all i = 0; i < N; i++ do
9:         MinDist = Double.MaxValue
10:        for all j = 0; j < cluster.length; j++ do
11:            dist = sqrt(computeDist(datapoint, centers[i]))
12:            if dist < minDist then
13:                minDis = dist
14:                index = i
15:            end if
16:        end for
17:        if cluster[i]! = index then
18:            updated = true
19:            cluster[i] = index
20:        end if
21:    end for
22:    if !updated then
23:        Break;
24:    end if
25:    for all i = 0; i < cluster.length; i++ do
26:        UpdateCluster(cluster[i]);
27:    end for
28: end for
```

FIGURE 4.21: K-Means Weka implementation.

An analysis of these different framework K-Means implementations is presented in contributions chapter (section 5.2).

4.3.2 Literature survey

Previous work done for Data Mining in GPU has been conducted. We will concentrate specifically in those regarding K-Means, since it was implemented in the SNIFF framework and many researches have been working with this algorithm.

A parallel K-Means implementation in [62], on GPU using CUDA technology was introduced. This implementation discussed specific design decisions to accelerate K-Means for the CUDA architecture. A recommendation that they gave is that when parallelizing an application to work in GPU, the characteristics of the architecture should be considered; for instance, GPU contains many cores compared to the CPU, which can allow to launch a large number of threads at the same time. In this architecture, the best is not to assign a thread to process each element, as there are millions of elements ready to be processed. For example, in the step of assignment of data points to each cluster, which is the time-dominant phase of the algorithm, the algorithm computes the Euclidean distance of each point that belongs to the set of centroids and assigns each point to the nearest cluster. To implement this in CUDA, in order to take into account the architecture of GPU, Reza Farivar et al. [62] proposed to assign the distance calculation of each data point to a single thread. Then, through all the initial cluster centroids, it calculates the distance of each data point per cluster at once, instead of computing each datapoint to its centroid in a loop; later on, finding the minimum distance for each datapoint to its closest centroid. The results of [62] for a GPU NVIDIA 8600GT were 13x times faster to a 3 GHz Intel Pentium.

Another authors such as [28], also implemented K-Means using GPU having significant better performance when compared to a fully SIMD (Single Instruction Multiple Data) optimized CPU implementation.

There has been some attempts to run GPU K-Means in large scale analytics. In [63] they did their research particularly in big data; their goal was to determine if GPU could be useful accelerators for data sets that cannot fit into memory. They used K-Means as the example and results proved to be very positive. For data sets smaller than GPU's onboard memory, their GPU version was 6-12x faster than their highly optimized CPU version running on an 8-core workstation. In the case of large data sets that cannot fit in GPU's memory, they showed a design which allows the computation on both CPU and GPU. With data transfer between them, the GPU version offered a great performance boost compared to the CPU version, achieving 9x performance in their test. The previous researchers extended their experiment in [29] to cluster a billion of datapoints using GPU. The results were still dramatic, having a 11x speed up for their GPU implementation compared to their highly optimized CPU version.

All these previous research works have shown great performance in K-Means using GPU compared to the CPU version for small and large datasets. Nevertheless, none of them explored the setup where each node in a cluster is equipped with a GPU, like in the SNIFF framework [1].

Chapter 5

CONTRIBUTIONS

In this chapter, good practices while working with data mining using GPU in general will be presented. These aspects need to be considered when working with GPU in order to execute data mining implementations efficiently. Also, the methodology used to optimize the K-Means implementation for SNIFF Framework will be shown, including all the steps and testings that were realized for finding the main bottleneck and the performance after solving this problem. Furthermore, another optimization of the K-Means algorithm using Thrust library when computing the total sum of distances with the datapoints and the centroids; as a result we got another improvement in the computation performance. Finally, an implementation of EM in SNIFF will be presented.

5.1 Data Mining Using GPU Requirements - Review

In this section, we will present some recommendations that are required to execute some data mining algorithms efficiently in GPU, as well as some kind of constructions that should be avoided.

In both implementations of EM and K-Means, computing the distance with the datapoints and the centroids was presented. As previously reported, to get a higher level of parallelism in K-Means [62], all the datapoints distance should be computed per centroid first. Then, calculate the minimum distance. e.g. when computing efficiently $d^2(x_i, m_j)$ in GPU. The algorithm passed through a series of parallel transformations, such as computing the difference of each datapoint per centroid and later on the square of each datapoint. The third step was to sum all the distances, which was not straightforward at the beginning, but it was possible to do it by using Reduce_by_key Thrust function, based on all the datapoints distances which will be further explained in the optimizations section. All these steps were efficiently executed in the K-Means algorithm, which were foundations to implement them in the EM algorithm, assuring great computations.

Another recommendation is the use of thrust::transformation having an own user defined operation performing many steps at once, as shown with the SAXPY example in the Thrust section, which is executed for each datapoint. For example, in the Naive Bayes implementation to compute the normal distribution $P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$ for each datapoint, it was possible to avoid all this complexity by using the transformation algorithm with a user defined function receiving the mean, variance and the datapoint as parameters. For computing the Maximum Likelihood in Expectation Maximization, the same parallel transformation was used for each datapoint, e.g. $P(x) = e^{-(x-\mu)^2/2\sigma^2}$ as presented in (2.1.6.5).

A construction that must be avoided is, in implementations of K-Means and EM, to always try to maximize the number of threads to execute, as previously mentioned. Launching many kernels, like computing one datapoint for all the centroids like in CPU implementations, can lead to a very bad performance due to the fact that launching kernels are very expensive operations [64].

In summary, Thrust library can significantly bring many benefits when implemented for data mining algorithms, since the structures presented brought high speed performance for our implementations; however, if the algorithms are not implemented exploiting the parallelism that can be gained from the GPU, then the resulting performance will be terrible.

5.2 Data Mining Frameworks K-Means implementations analysis

After going through all the K-Means implementations of the different frameworks, we came up with the following conclusions:

- Mahout K-Means, which is the only distributed implementation from the three frameworks studied, did not present any significant steps in the implementation which could add up significant performance. To the contrary, we noticed that it was doing two additional steps, which are not considered in SNIFF, that can make it slower, since it requires more computation. One step is doing a square root when computing the distance, which is not necessary [65]. Also, it is normalizing each distance to a probability where the closest distance contains higher probability (the closest centroid).

- R K-Means following the Lloyd implementation is quite similar to the implementation done in SNIFF. Since it is not doing the square root when computing the distance for each data point, not adding extra computation. Another difference is that it does not have a convergence threshold: it only stops when a maximum number of iterations has been reached or if there was no relocation of the clusters.
- Weka K-Means implementation based on [61] was quite similar to the R K-Means implementation, just doing the square root like in Mahout when computing the distance.

In summary, the implementations were quite similar to the K-Means GPU implementation in SNIFF framework. We did not take into account the possibility of significant improvements in the implementation of these frameworks and we decided to consider an analytic approach to find the main bottlenecks responsible for this slow performance, as shown in our methodology.

5.3 Methodology

In order to locate the parts of the system to optimize, we decided to use an analytic approach [66] that consisted in studying the system by its elementary elements, checking in detail each of those elements and the interaction between them, to finally find the parts that need to be optimized. Finding the main bottleneck was done through a series of steps, as shown in the following flowchart:

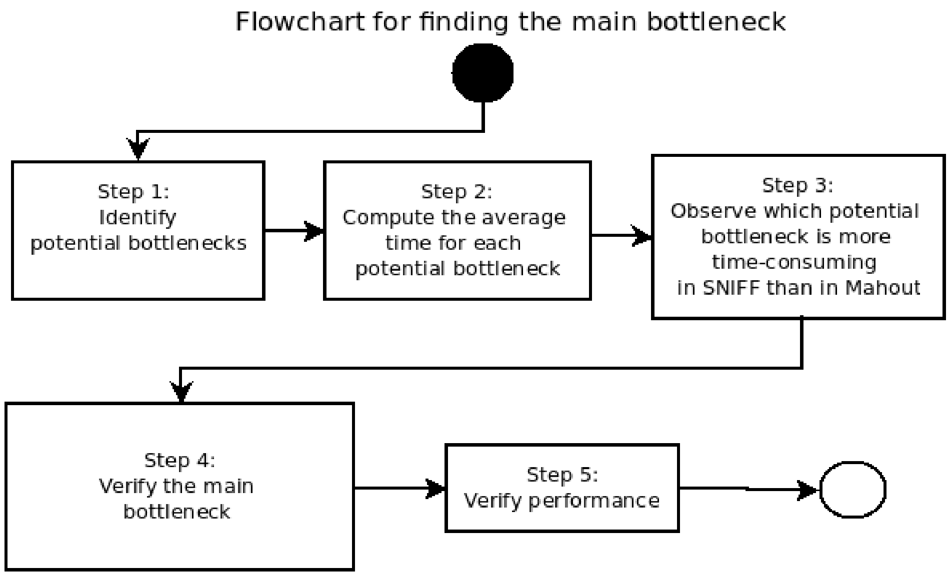

FIGURE 5.1: Flowchart with all the steps to identify potential bottlenecks

The steps in the chart are as follow:

- Step 1: Identify potential bottlenecks by comparing both implementations, Mahout and SNIFF.
- Step 2: Find which part is taking most of the time by computing the average time and comparing.
- Step 3: Observe which potential bottleneck is more time-consuming in SNIFF than in Mahout. For this step, we isolate the part of the implementation computing the distances in GPU and compare the average time to compute one centroid with one datapoint with large number of iterations.
- Step 4: After the findings of step 3, verify where the main bottleneck started in the implementation.
- Step 5: Verify the performance.

SNIFF and Mahout [57] use different formats for computing the data. In the case of SNIFF, the dataset is in text format. In the case of Mahout [57], since it only accepts SequenceFile format (A file containing key/value pairs), it was necessary to make a conversion into that format using a Java application, having as input for this application the dataset in text format and output in key/value pairs. An example of the format for both cases is presented in the following table:

Table 5.1 Example of the data formats

Text Format	SequenceFile Format (Key/Value)
1.0,162.0,4528.0,1.0,1.0,1.0,1.0 ...	0 / {0:1.0,1:162.0,2:4528.0,3:1.0,4:1.0,5:1.0,6:1.0 ...}

All tests were performed with EURA NOVA's computer, which contains 4 physical machines equipped with Intel i5 3550, 16 Gb RAM and Gigabit interconnect. The GPUs used for testing include the Nvidia GTX Titan (6 Gb device memory) and the Nvidia Tesla K20 (5 Gb) device memory [1].

Step 1: Identify potential bottlenecks

In the case of SNIFF, we tested the performance of the GPU implementation on three nodes, using different sizes of the KDD Cup 1999 Dataset (Appendix A), when testing K-Means we used 23 centroids and 32 dimensions. In Mahout, it was performed with the same dataset albeit poured to the SequenceFile format.

Step 2: Compute the average time for each potential bottleneck

After doing many tests, we discovered that both steps:

- Computing the distances
- Assigning the datapoints to the centroids

were taking most of the time, especially for SNIFF, since these tasks involve intensive computation, particularly computing the distances. The results of the test can be seen in the following figure:

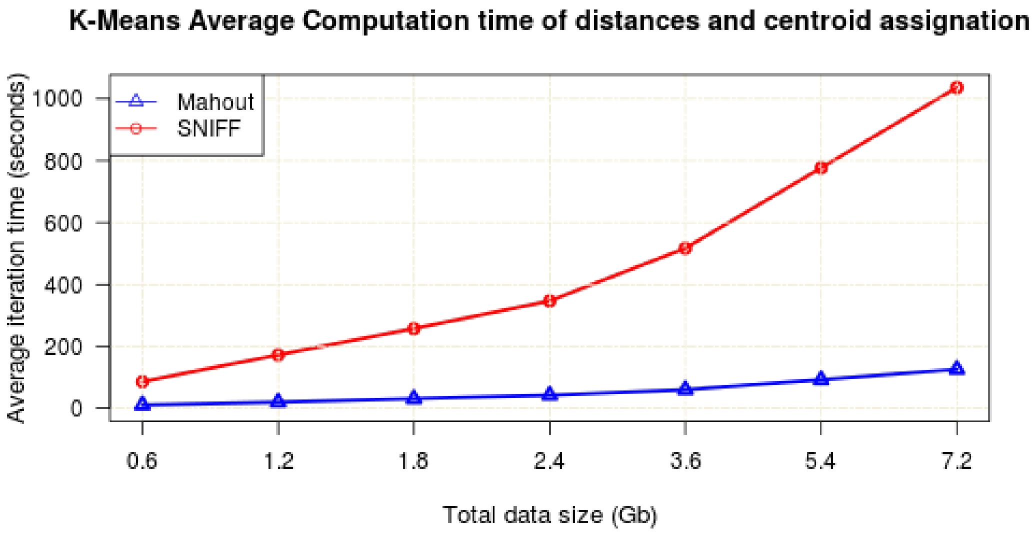

FIGURE 5.2: Average time spent for one iteration when computing all the distances with the datapoints and the centroids, and assigning the datapoints to the clusters on a GPU Node (Titan) in SNIFF and in one node of Mahout CPU.

Step 3: Observe which potential bottleneck is more time-consuming in SNIFF than in Mahout

We concluded that the biggest bottleneck in SNIFF was happening when it was computing the distances and assigning the datapoints to the centroids for both implementations. Doing this step was taking nearly 80 % of the average iteration time as illustrated in figure 5.3.

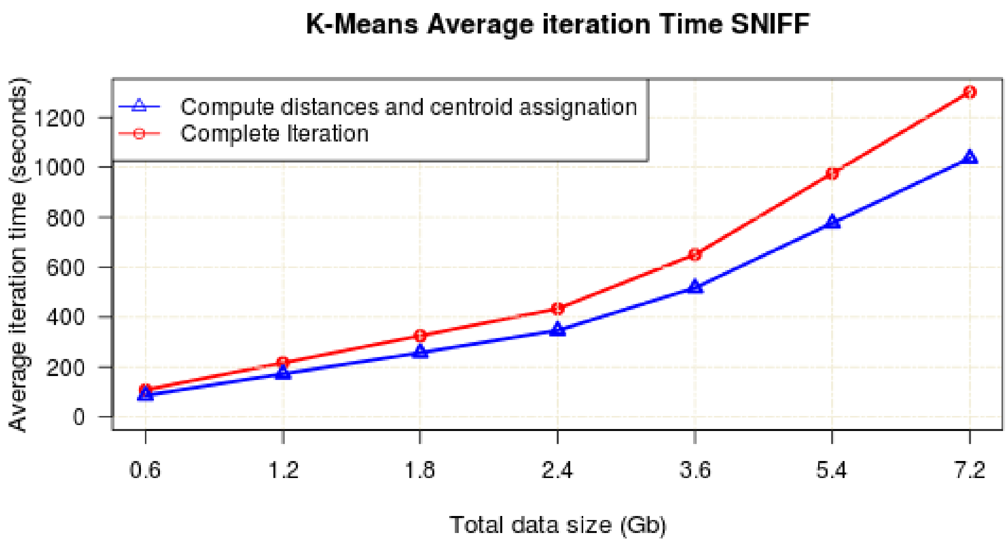

FIGURE 5.3: Comparison of the average time spent in one iteration on a GPU Node (Titan) and with the total time spent when computing the distances and the assignation of the datapoints to their centroids.

When the bottleneck was spotted, we looked for the cause of this huge difference in performance between SNIFF and Mahout. In Mahout, the steps of the algorithm (presented in related work) were inspected. It turned out that Mahout's K-Means implementation contained more computations, such as calculating a probability for each datapoint belonging to the cluster and doing a square root when computing the distance for each of the datapoint, two operations not performed on SNIFF. So in SNIFF there was a certain problem, since these steps add more computations that are not necessary such as the square root [65]. Even with all these steps, the performance for Mahout's K-Means implementation was superior by a great margin.

Isolate the step of computing the distances

In this case, we decided to totally isolate this step from the SNIFF implementation in order to focus in the procedure that computes the points and not in the way to load the points from disk. To later compute them in the case of SNIFF and find the potential main reason causing this bad performance.

In such manner, we selected just two vectors from dataset, one representing the datapoint and the other one representing the centroid. Then we compared a million times the distances between the two. We made the observations for both implementations and the results were quite surprising: the isolated implementation was nearly 35x faster than computing the distances in Mahout for 10,000,000 iterations, as shown in the following graph:

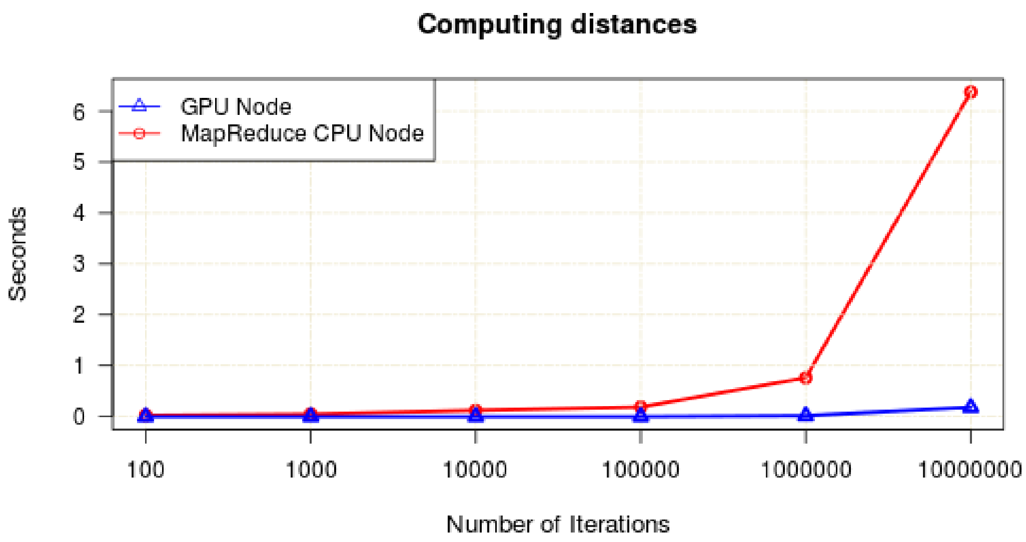

FIGURE 5.4: Comparison of the average time spent when computing distance of one centroid with one datapoint using 100, 1000, 10,000, 100,000, 1,000,000 and 10,000,000 iterations for a GPU Node (Titan) and in one CPU node when running Mahout K-Means.

Step 4: Verify the main bottleneck

At this time, we considered to integrate this implementation inside SNIFF in order to test the performance. We observed that it was receiving very poor performance at doing the same step. We continued testing this function in all the framework in order to know when this bad performance started and we noticed that, even at the beginning of the framework, it was causing this bottleneck. So, one of the only things left was to see the compilation parameters and, by using different parameters for compilation, we noticed that the parameter that was creating debugging information was the one causing this serious bad performance [67].

Correct the parameter

Below is a table with the description of the parameters included when compiling the project [68]:

OPTION	DESCRIPTION
nvcc	The CUDA compiler driver.
-O	Specify optimization level for host code.
-o	Specify name and location of the output file.
-g	Generate debug information for host code (CPU).
-G	Generate debug information for device code (GPU).
-odir	Specify the directory of the output file.
-M	Generate a dependency file that can be included in a makefile.
-arch	Specify the virtual NVIDIA GPU architectures to compile for.
-code	Specify the actual GPU archicture to compile for.
-gencode	Specifies a tuple of virtual and GPU architectures to target.

In order to create debug information in NVIDIA CUDA [18], it is necessary to use the parameter -g and -G to the nvcc compiler driver, but these options, apart of including debugging information, are also disabling most of the compiler optimizations [67]. So, using the -G parameter when compiling with NVCC will disable most of the compiler optimizations that are done in device code.

The following table presents an example of compilation:

Disabling the compiler optimizations	Enabling Compiler optimizations
/usr/local/cuda-5.5/bin/nvcc -O3 -gencode arch=compute_30,code=sm_30 -g -G -odir ”src/deviceFunctions” -M -o ”src/deviceFunctions/deviceFunctions.d” ”../src/deviceFunctions/deviceFunctions.cu”	/usr/local/cuda-5.5/bin/nvcc -gencode arch=compute_30,code=sm_30 -odir ”src/deviceFunctions” -M -o ”src/deviceFunctions/deviceFunctions.d” ”../src/deviceFunctions/deviceFunctions.cu”

Step 5: Verify performance

After enabling the compilation optimizations, we observed a huge improvement in performance for SNIFF. The result in the case of the comparison between SNIFF and Mahout went from 9x slower, to 10x faster for the NEW SNIFF than Mahout when compared with 7.2 Gb dataset, as we can see in the following figure:

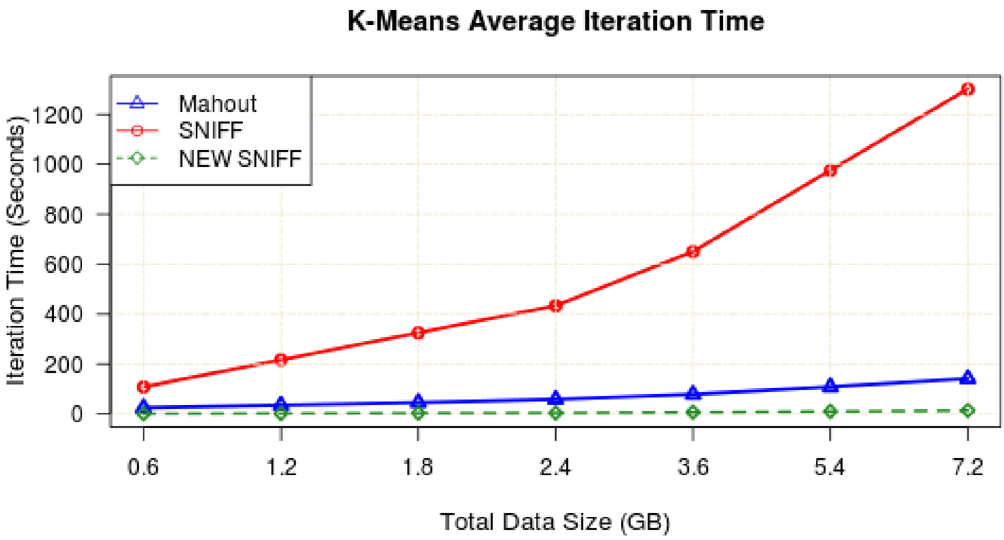

FIGURE 5.5: Average time spent for the computation on a GPU node (Titan) and on the Mahout for one iteration time disabling and enabling compiler optimizations for SNIFF. NEWSNIFF has more than 90x speed up performance compared to SNIFF, and 10x compared to Mahout.

Although it was extremely challenging to find the problem, the simple solution of adjusting the parameter actually solved the main problem of the thesis assignment. Nevertheless, we decided to further improve the performance.

5.4 Further Optimizations

In this section, we will present another optimization that was done in SNIFF GPU K-Means. Here, by changing a function and computing the total distance of the datapoints with the centroids, it was possible to obtain a speed up performance of the K-Means implementation by a factor of 1.5x when testing 7.2 Gb in 3 GPU nodes.

Optimization for computing the total distance of the datapoints with the centroids using Thrust

In order to obtain this increase in the speed performance, we will present two different approaches to compute the total distance of the datapoints with its nearest centroids. The Reduce_by_key algorithm, explained in Thrust section 2.1.6.7, can be used to compute the total distance. The one implemented in SNIFF required to create a vector for the input keys and the other one does not allocate memory in a vector, thus doing the computations on the fly. As a result, the new implementation is a faster algorithm with an equal output for both implementations.

In order to explain these two approaches, we will provide:

1. An example.
2. The input and output vectors that are required to compute Reduce_by_key.
3. The structure of the code of Reduce_by_key.
4. The previous implementation.
5. The new implementation.

1 - Example for computing total distance

The following example is used to explain these two different implementations of Reduce_by_key. Suppose we have the following 4 datapoints index of 3 dimensions, in which every dimension corresponds to the squared distance computed of the centroid with the datapoint in the context of K-Means. As shown in table 5.2.

Table 5.2 Example of the data

Datapoint Index	Dimension Values
1	10 20 30
2	10 30 40
3	30 40 50
4	20 20 30

Then, we want to compute the sum of the values for each datapoint. The table 5.3 summarizes the result of the sum of the dimensions. Note that for the datapoint 1, the value is 60. This value was obtained by the summed of the 3 dimensions: 10,20,30.

Table 5.3 Result of the sum of the values for each datapoint

Datapoint Index	Result of the sum of the dimension values
1	60
2	80
3	120
4	70

In order to do so in GPU, Thrust library contains data structures to stored the data on a Vector of 1 dimension to do the computations (as explained in Thrust section 2.1.6). Since each datapoint can contain many dimensions, as presented in table 5.2, it was necessary to do the corresponding sum for each datapoint. To perform this task, we used the Reduce_by_key algorithm and compute the final values. These final values represent the total distance of the datapoints with a cluster centroid. As displayed in table 5.3.

2 - Input and output vectors for Reduce_by_key

As we saw in Reduce_by_key section, it is necessary to use two vectors as input and two vectors for the output. In our example, the first input vector corresponds to the keys, which are the datapoint index with repetitions of the length of the dimensions to do the corresponding sum of the dimension values. The second vector values contain all the dimension values of the datapoints. These two vectors inputs are shown in the table 5.4.

Table 5.4 The input of Reduce_by_key are two vectors in 1 dimension: the vector keys and the vector values.

Vector Keys	1	1	1	2	2	2	3	3	3	4	4	4
Vector Values	10	20	30	10	30	40	30	40	50	20	20	30

After executing Reduce_by_key, it will produce as output two vectors: The vector keys, where same keys are grouped, and the vector values with the sum per group of vector keys; as demonstrated in table 5.5.

Table 5.5 The output of Reduce_by_key are two vectors in 1 dimension: the keys and the values.

Vector Keys	1	2	3	4
Vector Values	60	80	120	70

3 -Structure of the code for Reduce_by_key

As seen in Reduce_by_key section 2.1.6.7. In order to execute the algorithm, the structure of the code in listing 5.1, requires 5 parameters in the following order:

1. The input keys vector, which holds the values of the keys index repetition. e.g. (1,1,1,2,2,2,3,3,3,4,4,4.) as displayed in table 5.4.
2. The input keys vector termination, which is used to terminate the algorithm. e.g. for our example, 12, which is the size of the input vector.
3. The input values vector, which contains the values that will sum according to the key index repetition. For instance, (10,20,30,10,30,40,30,40,50,20,20,30), as presented in table 5.4.
4. The output keys vector, which holds the keys of grouping the same keys of input keys vector. e.g. (1,2,3,4) as shown in table 5.5.
5. The output values vector, which contains the sum of the values by grouping the keys of input values vector. e.g. (60,80,120,70) as presented in table 5.5.

The following listing 5.1 is the structure of the code of Reduce_by_key with the 5 parameters.

```
thrust::reduce_by_key(
        KeysInput.begin(),
        KeysInput.end(),
        ValuesInput.begin(),
        KeysOutput.begin(),
        ValuesOutput.begin()),
)
```

Listing 5.1: "Reduce_by_key structure"

Now, we will present the two approaches. The first one is the previous implementation of Reduce_by_key and the second one is the new implementation of Reduce_by_key.

4 - Previous implementation of Reduce_by_key

Next, an example of how it was computed in our previous implementation of Reduce_by_key in SNIFF K-Means will be presented, where we only have the vector with the values of the datapoints. As displayed in the table 5.6.

Table 5.6 The vectors with all the dimension values in 1 dimension.

Vector Values	10	20	30	10	30	40	30	40	50	20	20	30

Then, to perform the sum in Reduce_by_key algorithm in our previous implementation, we used an input vector holding the Keys. So we applied other algorithms from Thrust, explained in section 2.1.6, to create this input keys vector. One of them is thrust::counting_iterator used to generate the keys from the first datapoint to the last number of datapoint. For our example, it corresponds from 1 to 4. As presented by table 5.7.

Table 5.7 The vector with the keys.

Vector Keys	1	2	3	4

Later, we used Repeated_range thrust algorithm to repeat the same keys for each datapoint with the centroid size in order to sum the right number of dimensions. For our example with 4 datapoints with 3 values each, we have in table 5.8 the following vector keys:

Table 5.8 The input vector with all the keys.

Vector Keys	1	1	1	2	2	2	3	3	3	4	4	4

To create the vector keys of the table 5.8. SNIFF K-Means used structures such as counting_iterator and repeated_range. Counting_iterator is used to create the values from 1-4 as shown in table 5.7 and repeate_range create the values as presented in table 5.8. As displayed in the 5.2 listing.

```
//Start key Index with index 1.
thrust::counting_iterator<int> first(1);
//The last number of the datapoint index. For our example 4 datapoints.
thrust::counting_iterator<int> last = first + 4;
//Repeat the values for each key by the number of dimensions
//of the centroid, for this example 3.
repeated_range<thrust::counting_iterator<int> > keys(first, last, 3);
```

Listing 5.2: "Creating the key vector"

Finally, in the previous implementation of SNIFF K-Means, we get the the two vectors output displayed in table 5.5, by using the two input vectors (keys and values) for the Reduce_by_key function.

5 - New implementation of Reduce_by_key

For an optimization, we noticed, from a Thrust example called sum_rows [69], that it is not necessary to create the input key vector. It can be implemented with a different alternative. This alternative uses a function called linear_index_to_row_index along the Reduce_by_key to produce the input key vector on the fly, instead of creating the vector and allocating memory as in Listing 5.2.

By using different parallel algorithms to create the keys, such as counting_iterator and a function linear_index_to_row_index, we were able to create the respective keys vectors and reduce it to get the final distance computation per datapoint. In Appendix B Source code 2, the implementation of the new approach is presented.

Finally, the output vector was exactly the same as the one presented in table 5.5, containing the values of the distance for each datapoint.

The main difference, between the previous Reduce_by_key implementation and the optimized one with Reduce_by_key along linear_index_to_row_index, is that the first one requires to store the keys to compute Reduce_by_key. The second difference is computing the distance for each datapoint on the fly or by row and not allocating this key vector. Since the GPU performance is memory bound, by not allocating the vector keys in the memory, the speed performance increases [70].

Verify performance of the two implementations of Reduce_by_key

Using this new approach, SNIFF K-Means implementation increased its performance, resulting for the datasize of 7.2 Gb with a computing performance 1.5x faster than computing a K-Means iteration with the compiler optimizations enabled, as seen in the following graph:

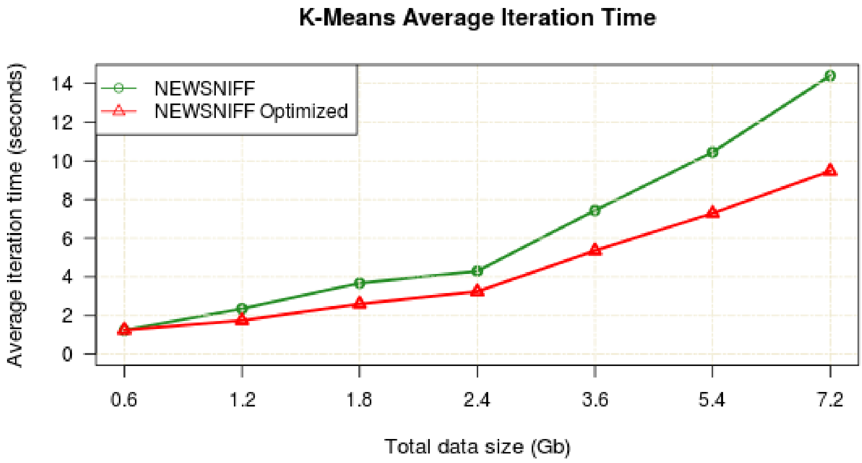

Figure 5.6: Average time spent for the computation on a GPU node (Titan) and on new SNIFF for one iteration time after using new implementation of Reduce_by_key.

Based on the results of the previous figure, it is clear that working with the new implementation of Reduce_by_key provides a considerable gain in the speed performance.

5.5 Expectation Maximization Implementation in SNIFF

Expectation Maximization (EM) was implemented to test the new insights with a related implementation such as the SNIFF K-Means. It was based on the algorithm shown in [11] and explained in section 4.1.2, in which all the steps were implemented, just with the exception that the variances of the clusters are recalculated based on the values of the new means for the following iterations.

The implementation based on [11] (section 4.1.2) of EM following the SNIFF model was very straightforward. Both intensive computations, such as the Expectation step and Maximization step, were executed in the workers equipped with GPU and finally, after computing each part in every node in the maximization step, we sent all the values of the maximization step to the combiner to compute the global mean. Nevertheless, in order to compute the global variance, it is necessary to compute the global mean first and later to broadcast it to each of the GPU nodes and compute the global variance before the next iteration. EM stop iterating until it converges or reaches a maximum number of iterations, if not, EM will keep iterating using the global means and global variances for computing the next iterations.

Algorithm 8 Distributed Expectation Maximization - Worker Routine

Read data T from local data partition in GPU memory
Set $active = true$
while $active$ **do**
    $centroids$ ←Receive k centroids $\{m_j\}$ from combiner
    $variances$ ←Receive σ Variances $\{v_j\}$ from combiner
    for all centroids m_j **do**
        $D_{ij} \leftarrow$ Compute all $exp[-(x_i - m_i^t)^2/2\sigma^2)]$
    end for
    $ct'_j \leftarrow$ Increment count number based on $exp[-(x_i - m_j^t)^2/2\sigma^2)]$ per x_i for each index j
    $E(Z_{ij}) \leftarrow D_{ij}$ divided by ct'_j
    for all centroids m_j **do**
        $\{m'_j\} \leftarrow$ compute all $x_i * E(Z_{ij})$
    end for
    Send $\{m'_j\}$, ct'_j to $Combiner$
    $active \leftarrow$ get $active$ response from $Combiner$
    Receive m_j from combiner
    for all centroids m_j **do**
        $\{v'_j\} \leftarrow$ compute all $(x_i - m_j^t)^2 * E(Z_{ij})$
    end for
    Send $\{v'_j\}$, ct'_j to $Combiner$
    $active \leftarrow$ get $active$ response from $Combiner$
end while

FIGURE 5.7: Distributed Expectation Maximization Worker Routine.

Algorithm 9 Distributed Expectation Maximization - Combiner Routine

Generate k initial centroids $\{m_j\}$
Generate σ initial variances $\{v_j\}$
do
    Send $\{m_j\}$, $\{v_j\}$ and *active* to the w workers
    Receive model parts $\{m_j\}'$, ct'_j from all w workers
    $ct_j \leftarrow \sum ct'_j$
    $m_j \leftarrow \frac{\sum m'_j}{ct_j}$
    Send $\{m_j\}$ and *active* to the workers
    Receive model parts $\{v_j\}'$ from all w workers
    $v_j \leftarrow \frac{\sum v'_j}{ct_j}$
while $||\Theta_k^{t+1} - \Theta^t|| > threshold$
Send $active \leftarrow false$ to the w workers

FIGURE 5.8: Distributed Expectation Maximization Combiner Routine.

We did some experiments executing one iteration of EM using the same set up as the K-Means GPU implementation. Computing both the global means and the global variances, the speed was significant; computing 7.2 Gb of data in just 27 seconds in 3 GPU nodes. The following figure shows the computation from 0.6 Gb to 7.2 Gb in 3 GPU nodes:

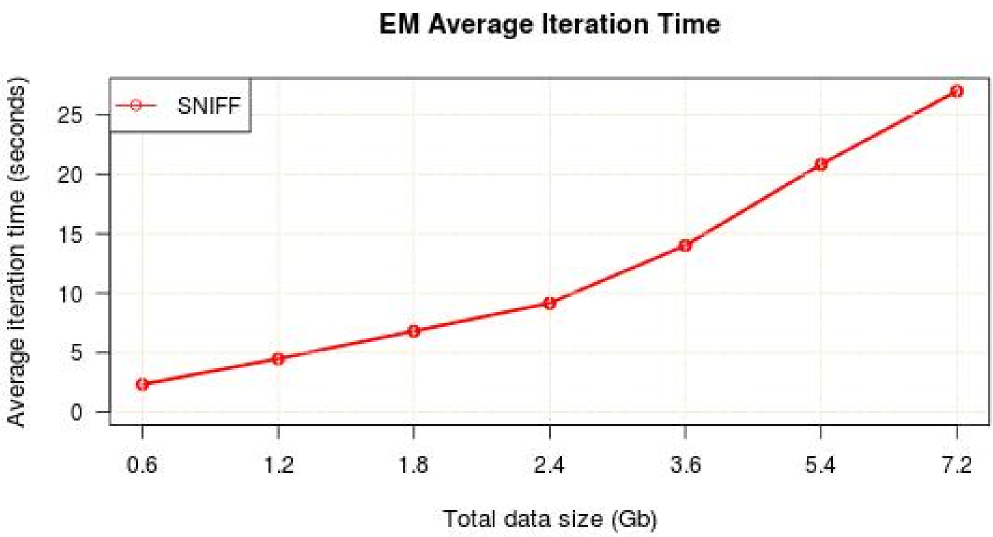

FIGURE 5.9: EM Average time for the computation on a GPU node (Titan) for one iteration

Chapter 6

Final Planning

The main objective of the thesis was to optimize the K-Means implementation in the SNIFF framework taking into account the optimizations of the different frameworks previously mentioned. At the end, the speed performance was affected by a simple parameter, as explained in contributions chapter. After solving the main objective, we decided to add another implementation to test new insights during the short amount of time remaining. We chose the Expectation Maximization technique for soft clustering, since it requires intensive computation. Also, it is similar to the previous K-Means implementation by EURA NOVA, so this new implementation was added to our initial planning.

Timeframe estimation for the Final Planning:

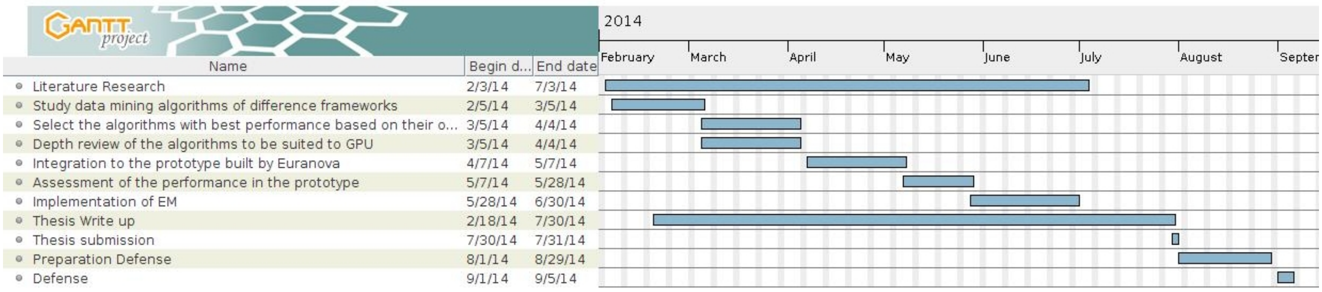

FIGURE 6.1: Final Planning

Chapter 7

CONCLUSIONS & FUTURE WORK

To answer the objective of this thesis we studied different framework implementations. Based on the previous implementations of K-Means, as presented in chapter 3, in Mahout there are extra steps which are not included in SNIFF. Because these steps make the computation performance slow, it was surprising that Mahout's implementation was actually faster than the SNIFF K-Means implementation. Due to the nature of R and Weka implementation, there is not any significant optimizations that could be adopted in the SNIFF framework. Based on these findings, we changed our approach to achieve our objective, as mentioned in the contributions chapter.

As mentioned in the contributions chapter, to increase the performance of K-Means implementation, we used an analytical approach. Through this, we isolated the step that was taking most of the computation from the implementation, which, in this case, was computing the distance. After all these steps, we were able to increase the performance of this implementation by a factor of 130 times, and 10 times faster than the Mahout CPU implementation. Nevertheless, we continued improving the performance by applying new structures to a parallel algorithm from Thrust Library called Reduce_by_key, achieving a 1.5 times performance after removing the compilation parameter causing the performance problems.

This significant improvement in performance should motivate to extend the application of the DFG model to clusters of nodes equipped with GPUs [1].

We also added a new data mining implementation into SNIFF framework, in order to make soft clustering. This implementation is Expectation Maximization. It is an iterative approach which allows to reuse some parts of the code from the K-Means implementation; but it required more computation since, after computing the global Means, it needed to broadcast the global means to each of the workers and compute locally the variances to finally get those parts and compute them in the combiner node.

In summary, the implementation of GPU K-Means was not in fact wrong or doing unnecessary steps. Getting rid of the debugging parameter did not only improve the K-Means implementation, but also any other implementation based on that framework; we achieved improvement implementation of that framework.

Also, it is recommended to check those functions which require intensive computation, since it can be tuned to get further optimizations such as using row_by_index function along reduce_by_key.

The comparison was only between the distributed version of Mahout and SNIFF GPU K-Means. Further distributed implementations such as RHadoop (R using Hadoop) with K-Means was not compared in this period of time; it might be interesting to check whether it can have very different performance compared to SNIFF.

Due to the time, in the K-Means framework analysis, there is an absence of a qualitative comparison. It would be interesting to see, whether one implementation gives better quality than the other, in order to know if the implementations did not compromise the results.

In the case of EM, it was planed to compare the performance with Mahout, but up to our knowledge, there was not an application available. Also since it is a technique, it would be interesting to compare how it was implemented in other distributed framework. As well as testing if the EM implementation performs better than K-Means in retrieving the real clusters.

Appendix A

Dataset

The KDD Cup 1999 Dataset [71] was used for the experiments. Containing different levels of Gaussian noise added in order to inflate the total volume of data from 0.6Gb to 7.2Gb. The dataset contains TCP connection information simulating a U.S. Air Force LAN, each connection presents 41 features, which 32 are continuous and 9 discrete and contains a label either a normal or a network attack [1].

Appendix B

Source Code

1 - A cuda implementation retrieved from [45], in which it shows how to compute the square of a number in an array in parallel using GPU.

```
//Steps executed in the CPU such as memory allocation and launching the Kernel
    to be executed in the GPU.

#include <stdio.h>

int main(int argc, char ** argv) {
        const int ARRAY_SIZE = 100;
        const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

        // generate the input array on the host
        float h_in[ARRAY_SIZE];
        for (int i = 0; i < ARRAY_SIZE; i++) {
                h_in[i] = float(i);
        }
        float h_out[ARRAY_SIZE];

        // declare GPU memory pointers
        float * d_in;
        float * d_out;

        // allocate GPU memory
        cudaMalloc((void**) &d_in, ARRAY_BYTES);
        cudaMalloc((void**) &d_out, ARRAY_BYTES)

// transfer the array to the GPU
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
```

```
// launch the kernel
square<<<1, ARRAY_SIZE>>>(d_out, d_in);

// copy back the result array to the CPU
cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

// print out the resulting array
for (int i =0; i < ARRAY_SIZE; i++) {
        printf("%f", h_out[i]);
        printf(((i % 4) != 3) ? "\t" : "\n");
}

cudaFree(d_in);
cudaFree(d_out);

return 0;
}

//The Kernel in which it will execute all the threads in parallel in a block.
    Squaring the number and storing in the array.

__global__ void square(float * d_out, float * d_in){
        int idx = threadIdx.x;
    float f = d_in[idx];
    d_out[idx] = f * f;
}
```

2 - The new implementation of Reduce_by_key used in SNIFF optimized the sum to compute the final distance computation between the dimensions of the centroid and the datapoint.

The goal of the following algorithm is to create the following sequence for input key vector (1,1,1,2,2,2,3,3,3,4,4,4) and make the sum of the input values vector on the fly by grouping the keys to save memory. The result is exactly the same output as the previous Reduce_by_key implementation.

- Line [2]: We want to sum all the values per rows. In this case, each row is a datapoint conformed of 3 dimensions. Then, instead of creating the keys Input vector, we have an operation called make_transform_iterator which will transform the values from the counting iterator and index_to_row_index, where make_transform_iterator will iterate given by the number in counting iterator (1) and number of times given by the second argument (3). In this case producing a set (1,1,1). Counting_iterator keeps increasing its value after a complete iteration over the elements of the row given by index_to_row_index. index_to_row_index contains the number of the row size that it has to compute, in this case 3 dimensions.
- Line [3]: The next make_transform_iterator specifies the size of each iteration and the termination of the algorithm, (4*3) → 4 rows with 3 dimensions for this case. Producing the following 4 sets for input Keys per row on the fly: (1,1,1), (2,2,2), (3,3,3), (4,4,4). For each set of same keys, the algorithm is making the sum on the fly and putting the result of the Values in the ValuesOutput vector.

The code of the Optimize version of Reduce_by_key is the following:

```
1 thrust::reduce_by_key(
2 thrust::make_transform_iterator(thrust::counting_iterator<int>(1),
      linear_index_to_row_index<int>(3)),
3 thrust::make_transform_iterator(thrust::counting_iterator<int>(1),
      linear_index_to_row_index<int>(3)) + (4*3),
4 //Values that will be summed according to the key index repetition.
5 ValuesInput.begin(),
6 //Vector containing the output keys.
7 KeysOutput.begin(),
8 //Vector containing the final values representing the distances.
9 ValuesOutput.begin())
```

Listing B.1: "Optimize version of Reduce_by_key"

Bibliography

[1] Nam-Luc Tran, Quentin Dugauthier, and Sabri Shkiri. A distributed data mining framework accelerated with graphics processing units. 2013.

[2] Geforce gtx titan, 2014. URL http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan.

[3] Apache Hadoop. What is apache hadoop? URL http://hadoop.apache.org/.

[4] Alan Tatourian. Nvidia gpu architecture and cuda programming environment. URL http://code.msdn.microsoft.com/windowsapps/NVIDIA-GPU-Architecture-45c11e6d.

[5] Universita degli Studi di Catania. An introduction to gpu programming. URL http://www.dmi.unict.it/~bilotta/gpgpu/ingv/.

[6] Cliff Woolley. Gpu optimization fundamentals, 2013. URL https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf.

[7] Martin Przyjaciel-Zablocki Thomas Hornung and Alexander Schätzle. Giant data: Mapreduce and hadoop, 2014. URL http://www.admin-magazine.com/HPC/Articles/MapReduce-and-Hadoop.

[8] Dattatrey Sindol. Introduction to mapreduce, 2014. URL http://www.mssqltips.com/sqlservertip/3222/big-data-basics--part-5--introduction-to-mapreduce/.

[9] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in action*. Manning Publications Co., 2011.

[10] Victor Lavrenko and Charles Sutton. Mixture models and em, 2011. URL http://www.inf.ed.ac.uk/teaching/courses/iaml/2011/slides/em.pdf.

[11] Sara Nasser, Rawan Alkhaldi, and Gregory Vert. A modified fuzzy k-means clustering using expectation maximization. 1996.

[12] Esteves Rui Maximo, Chunming Rong, and Pais Rui. K-means clustering in the cloud - a mahout test. 2011.

[13] Data Mining Algorithms In R Clustering K-Means. K-means, April 2014. URL http://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Clustering/K-Means.

[14] Internet 2012 in numbers, January 2013. URL http://royal.pingdom.com/2013/01/16/internet-2012-in-numbers/.

[15] Neelamadhab Padhy, Dr. Pragnyaban Mishra, and Rasmita Panigrahi. The survey of data mining applications and future scope. 2012.

[16] Jeff Tyson and Tracy V. Wilson. How graphics cards work, 2011. URL http://computer.howstuffworks.com/graphics-card1.htm.

[17] Marshall Brain. How microprocessors work, 2011. URL http://computer.howstuffworks.com/microprocessor.htm.

[18] NVidia. What is cuda? URL http://www.nvidia.com/object/cuda_home_new.html.

[19] Khronos Group. The open standard for parallel programming of heterogeneous systems. URL https://www.khronos.org/opencl/.

[20] Wen mei W. Hwu. *GPU Computing Gems Emerald Edition, Thrust: A Productivity-Oriented Library for CUDA*.

[21] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. 2010.

[22] R. O. Duda and P. E. Hart. Pattern classification and scene analysis. 1973.

[23] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. 1979.

[24] Apache Software Foundation Mahout. Mahout k-means, April 2014. URL https://mahout.apache.org/users/clustering/k-means-clustering.html.

[25] Michael Saecker and Volker Markl. Big data analytics on modern hardware architectures: A techonology survey. 2012.

[26] Christian Bohm, Robert Noll, Claudia Plant, Bianca Wackersreuther, and Andrew Zherdin. Data mining using graphics processing units. 2009.

[27] Martin Ester, Hans peter Kriegel, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. 1996.

[28] Mario Zechner and Michael Granitzer. Accelerating k-means on the graphics processor via cuda. 2009.

[29] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering billion of data points using gpus. 2009.

[30] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, and Philip Yang Yang. Parallel data mining on graphics processors. 2006.

[31] Feng Cao, Anthony K. H. Tung, and Aoying Zhou. Scalable clustering using graphics processors. 2006.

[32] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. December 2009.

[33] Nam-Luc Tran, Sabri Skhiri, Arthur Lesuisse, and Esteban Zimanyi. Arom: Processing big data with data flow graphs and functional programming. 2012.

[34] Mihai Budiu Michael Isard, Andrew Birrell Yuan Yu, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. 2007.

[35] The R Project for Statistical Computing. R, April 2014. URL http://www.r-project.org/.

[36] Weka, April 2014. URL http://www.cs.waikato.ac.nz/ml/weka.

[37] Nam-Luc Tran and Sabri Skhiri. Master thesis proposals 2013-2014 faculty of science engineering eura nova, revision 3, November 2013. URL http://it4bi.univ-tours.fr/medias/images/Master-Thesis-2013-2014_EN_SLW1-1.pdf.

[38] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the em algorithm. 1977.

[39] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. 2010.

[40] What is gpu accelerated computing?, MAY 2014. URL http://www.nvidia.com/object/what-is-gpu-computing.html.

[41] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. 2004.

[42] Byungil Jeong and Greg Abram. 8 things you should know about gpgpu technology, 2010. URL https://www.tacc.utexas.edu/news/feature-stories/2010/8-things-you-should-know-about-gpgpu-technology.

[43] Allan Gottlieb and George S. Almasi. Highly parallel computing. 1989.

[44] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors A Hands-on Approach*. Morgan Kaufmann, 2013.

[45] David Luebke and John Owens. Intro to parallel programming, 2013. URL https://www.udacity.com/course/viewer#!/c-cs344/l-55120467/m-67074291.

[46] NVidia. Cuda toolkit documentation - thrust, 2014. URL http://docs.nvidia.com/cuda/thrust/.

[47] Thrust group. Transformations, 2013. URL http://thrust.github.io/doc/group__transformations.html.

[48] Alex Holmes. *HADOOP in Practice*. MANNY, 2012.

[49] JeffreyDean and SanjayGhemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[50] Dr. Daisy Zhe Wang. Introduction to data science/data intensive, 2014. URL http://cise.ufl.edu/class/cis6930sp14ids/PageRank.pdf.

[51] Max Bramer. *Principles of Data Mining*. Springer, 2013.

[52] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. 1996.

[53] Daniel T. Larose. *Discovering Knowledge in Data: An Introduction to Data Mining*. WILEY, 2005.

[54] Data mining algorithms in r/clustering/expectation maximization (em), May 2014. URL http://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Clustering/Expectation_Maximization_%28EM%29.

[55] Yanchang Zhao. R and data mining: Examples and case studies, April 2013.

[56] Yanchang Zhao. R reference card for data mining, May 2014. URL http://www.rdatamining.com/docs.

[57] Apache Software Foundation Mahout. Mahout, April 2014. URL https://mahout.apache.org/.

[58] I. Dhillon and D. Modha. A data-clustering algorithm on distributed memory multiprocessor. 2002.

[59] A.M. Middleton. Data-intensive technologies for cloud computing, 2010.

[60] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on map reduce. 2009.

[61] Sapna Jain, M Afshar Aalam, and M. N Doja. K-means clustering using weka interface. 2010.

[62] Reza Farivar, Daniel Rebolledo, Ellick chan, and Roy Campbel. A parallel implementation of k-means clustering on gpu. 2008.

[63] Ren Wu, Bin Zhang, and Meichun Hsu. Gpu-accelerated large scale analytics. 2009.

[64] Cuda kernel overhead, 2009.

[65] Distance approximations, May 2013. URL http://en.wikibooks.org/wiki/Algorithms/Distance_approximations.

[66] J. de Rosnay. Analytic vs. systemic approaches, February 1997. URL http://pespmc1.vub.ac.be/ANALSYST.html.

[67] Thrust, May 2014. URL http://www.roguewave.com/portals/0/products/totalview-family/totalview/docs/8.11/index.html#page/User_Guides/totalviewug-cuda-tutorial.31.02.html.

[68] Nicholas Wilt. *The CUDA Handbook A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013.

[69] Nathan Bell. Github thrust, 2011. URL https://github.com/thrust/thrust/blob/master/examples/sum_rows.cu.

[70] Lawrence Latif. Acceleware says most cuda applications are memory bound, 2013. URL http://www.theinquirer.net/inquirer/news/2255592/acceleware-says-most-cuda-applications-are-memory-bound.

[71] The UCI KDD Archive. Kdd cup 1999 data, October 1999. URL https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data.