Technische Universität Berlin

Master Thesis

Machine Learning from Streaming Data in Heterogeneous Computing Environments

Submitted to the Faculty IV, Electrical Engineering and Computer Science Database Systems and Information Management Group in partial fulfillment of the requirements for the degree of Master of Science in Computer Science as part of the Erasmus Mundus IT4BI programme at the Technische Universität Berlin

Yusuf Güven Toprakkiran

Matriculation #: 0376651

Supervisors: Prof. Dr. Volker Markl
Advisors: Viktor Rosenfeld, Jonas Traub

31/07/2016

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare to have written this thesis on my own and without forbidden help of others, using only the listed resources.

Datum	Yusuf Güven Toprakkiran

Contents

List of Figures vi

List of Listings viii

List of Tables viii

Acronyms ix

1. English Abstract 1

2. Deutscher Abstract 2

3. Introduction 3
3.1. Graphics Processors for General Purpose Computations 6
3.2. Data Stream Clustering 6
3.3. Motivation 8
3.4. Contributions 10
3.5. Thesis Outline 11

4. Background 12
4.1. Parallel Computing 12
4.2. Apache Flink 13
4.3. Heterogeneous Computing 14
4.4. OpenCL 15
4.4.1. Platform Model 16
4.4.2. Execution Model 16
4.4.3. Memory Model 17
4.4.4. Programming Model 18
4.5. Data Stream Processing 19
4.6. Data Stream Clustering and K-means Algorithm 20

5. Related Work 23

6. Implementation 26
6.1. Architecture Overview 26
6.2. Streaming Server 27
6.3. OpenCL Implementation 27
6.4. Apache Flink Implementation 30
6.5. Dataset Generation 32

7. Evaluation **34**
7.1. Experimental Setup and Datasets 34
7.2. Evaluation Strategy 36
7.3. Preliminary Experiments 38
7.3.1. Global Work Size 38
7.3.2. Data Type 40
7.4. Throughput Experiment Results 43
7.4.1. Window Size 43
7.4.2. Tuple Size 45
7.4.3. Number of Clusters 48
7.4.4. Window Slide Size 49
7.5. OpenCL and Flink Performance Comparison 52

8. Conclusion **59**

References **i**

Appendix **vii**

A. OpenCL Kernels **A.1**

B. Additional Results for OpenCL Experiments **A.3**
B.1. Cost of the Internal Steps in the Window Size Experiments A.4
B.2. Cost of the Internal Steps in the Tuple Size Experiments A.5
B.3. Cost of the Internal Steps in the Cluster Count Experiments A.6

C. Additional Results for Apache Flink Experiments **A.7**

D. Statistics **A.10**
D.1. Euclidean Distance A.10
D.2. Gaussian Distribution A.10

List of Figures

1. Clock rate and power consumption of Intel x86 microprocessors 3
2. Categories of concept drift applications 5
3. Platform model of OpenCL [47] 16
4. Structure of 2-dimensonal index space, taken from [58] 17
5. Memory model of OpenCL [7] 18
6. Overview of OpenCL and Flink streaming k-means implementations 26
7. Sliding window concept used in the experiments 27
8. Overview of the OpenCL application 28
9. Structure of the array which is used for keeping aggregation values of new centroids in the OpenCL kernel 30
10. Execution plan of the Flink program generated by Flink visualizer 31
11. Workflow inside of the streaming k-means window 32
12. Cost of the centroid update in host CPU for different global work sizes . . 39
13. Computational time for one window with different global work sizes in GPUs 39
14. Computational time for one window with different global work sizes in CPUs 40
15. The impacts of the data type in GPU devices on different phases in OpenCL application 41
16. The impact of the data type in CPU devices on different phases in OpenCL application 42
17. The impact of the data type in Flink on CPTW and throughput 43
18. Impact of the window size on throughput in OpenCL application 44
19. Impact of the window size on throughput in Flink 45
20. Impact of tuple size on throughput, in OpenCL application (lower tuple sizes) 46
21. Impact of tuple size on throughput, in OpenCL application (higher tuple sizes) 46
22. Impact of the tuple size on throughput in Apache Flink 47
23. Impact of the tuple size on throughput in OpenCL application 48
24. Impact of cluster count on throughput in Apache Flink 49
25. Impact of the slide size on throughput in OpenCL application 50
26. Impact of the slide size on throughput-2 in OpenCL application 50
27. Impact of the slide size on throughput in Flink 51
28. Impact of the slide size on throughput-2 in Flink 51
29. OpenCL vs Flink throughput comparison for window size experiments . . 53
30. OpenCL vs Flink throughput comparison for tuple size experiments (lower tuple sizes) 54
31. OpenCL vs Flink throughput comparison for tuple size experiments (higher tuple sizes) 54

32. OpenCL vs Flink throughput comparison for the number of clusters experiments. 55
33. OpenCL vs Flink throughput comparison for slide size experiments 56
34. OpenCL vs Flink throughput-2 comparison for slide size experiments . . . 56
35. Window preparation cost of different CPUs in OpenCL application A.4
36. Cost of the internal steps in OpenCL application, window size experiments A.4
37. Cost of the internal steps in OpenCL application, tuple size 2-16 . . . . . A.5
38. Cost of the internal steps in OpenCL application, tuple size 32-256 . . . . A.6
39. Cost of the internal steps in OpenCL application, number of cluster experiments A.7
40. Cost of the internal steps in Flink application, the window size experiments A.8
41. Cost of the internal steps in Flink application, the tuple size experiments . A.8
42. Cost of the internal steps in Flink application, the number of clusters experiments A.9
43. Cost of the internal steps in Flink application, window slide size experiments A.9

List of Listings

1. An example of scalar fuction in C 18
2. An example of data parallel OpenCL kernel 19
3. General algorithm of the OpenCL kernels 29
4. Data generator algorithm 33
5. OpenCL kernel-1 (four-elements vector type) A.1
6. OpenCL kernel-2 . . . . . A.2

List of Tables

1. Computing devices in the System-1 and System-2 35
2. Comparison of the OpenCL applications, *AMD Radeon R9 Fury* vs *AMD Opteron 6376* 53
3. Flink vs OpenCL (GPU, *AMD Radeon R9 Fury)* 57
4. Flink vs OpenCL (CPU, *AMD Opteron 6376*) 57

List of Acronyms

API Application Programming Interface
JVM Java Virtual Machine
GPU Graphics Processing Unit
CPU Centrol Processing Unit
GPGPU General-purpose Computing on Graphics Processing Units
ML Machine Learning
SIMD Single Instruction Multiple Data
MIMD Multiple Instruction Multiple Data
SIMT Single Instruction Multiple Thread
CTPW Computational Time Per Window
APU Accelerated Processing Unit
FPGA Field-programmable Gate Array

1. English Abstract

With the advent of many-core general-purpose processors (CPUs), the use of an increased number of cores has provided a certain speedup for algorithms that can be parallized. Nowadays, there are distributed and parallel data processing platforms, such as Apache Flink, which inherently makes use of parallel computing. On the other hand, graphics processors(GPUs) offers high performance solutions for certain problems thanks to their architecture that is suitable for massivelly data parallel computations. In the last decade, GPU computing has became popular also for general purpose applications. Although there are some drawbacks such as memory transfer latency, it has been proven that GPUs provide substantial speedup especially in computationally intensive problems thanks to their massively parallel computation capability. Nowadays, there are also heterogeneous computing platforms such as OpenCL which enables developers to write portable programs that can be executed in parallel in a range of processors such as CPUs and GPUs while providing certain abstractions that simplify parallel programming across different computing devices.

Streaming k-means is an unsupervised online learning algorithm which is an adaptation of batch k-means algorithm which is still one of the most commonly used algorithms due to its simplicity, efficiency and empirical success. In this thesis, we initially implement sliding window based streaming k-means algorithm in OpenCL and Apache Flink, and give an overview regarding the impact of the window size, the tuple size, the number of clusters and the window slide size on system throughput in two CPUs and three GPUs. We achieve higher throughput than Flink in our OpenCL application. Besides, we show that GPUs still produce higher throughput than many-core CPUs. However, the difference between the performances of OpenCL applications where the computational intensive step is executed in CPUs and GPU is reduced in modern architectures. Furthermore, the modern many-core CPUs can occasionally show competitive performance with GPUs in particular when our streaming k-means algorithm is used.

2. Deutscher Abstract

Mit dem Aufkommen der Mehrkernallzweckprozessoren (CPUs), hat die Verwendung einer erhöhten Anzahl von Kernen eine gewisse Beschleunigung für parallelisierbare Algorithmen erzielt. Heutzutage gibt es verteilte und parallele Datenverarbeitungsplattformen wie Apache Flink, die von Natur aus Verwendung von paralleler Verarbeitung machen. Andererseits, bieten Grafikprozessoren (GPUs), dank ihrer Architektur, die für massiv parallele Berechnungen geeignet ist, Hochleistungslösungen für bestimmte Probleme. Im letzten Jahrzeit wurde GPU-Computing auch für allgemeine Anwendungen beliebt. Trotz einiger Nachteile, wie beispielsweise Speicherübertragungslatenz, hat sich erwiesen, dass GPUs erhebliche Beschleunigung, insbesondere in rechenintensiven Problemen erreichen können. Heutzutage gibt es auch heterogene Computing-Plattformen wie OpenCL, die es ermöglichen portable Programme auf einer Reihe von Prozessoren wie CPUs und GPUs unter Bereitstellung bestimmter Abstraktionen parallel auszuführen und somit die parallele Programmierung auf verschiedenen Geräten zu vereinfachen.

Streaming-k-means ist ein unüberwachter Online-Lernalgorithmus, der eine Anpassung des Batch k-means Algorithmus darstellt. In dieser Arbeit setzen wir zunächst einen Window- basierten Streaming-k-means Algorithmus in OpenCL und Apache Flink um. Wir geben einen Überblick über die Auswirkungen der Windowgröße, der Tupelgröße, der Anzahl an Clustern und der Windowschrittgröße auf den Systemdurchsatz in zwei CPUs und drei GPUs. Wir erreichen einen höheren Durchsatz als Flink in unserer OpenCL Anwendung. Außerdem zeigen wir, dass GPUs noch höheren Durchsatz als Many-Core-CPUs produzieren. Wenn Jedoch wird der Unterschied zwischen den Leistungen der OpenCL-Anwendungen die rechenintensive Schritte auf CPUs und GPUs ausführen in modernen Architekturen reduziert. Darüber hinaus können insbesondere moderne Many-Core CPUs gelegentlich wettbewerbsfähige Leistungen zu GPUs zeigen, wenn unser Streaming-k-Means-Algorithmus verwendet wird.

3. Introduction

In recent decades, the computer industry has commonly benefited from technological advances in the semiconductor industry and made use of the rapid rise in transistor speed. The performance of transistors has been doubled every 18 months and this increased the performance of computers while sustaining the sequential programming model [12]. However, as this trend has been disrupted by the power limits of processors, research started to focus on providing enhanced performance with lower frequencies as well as lower energy consumption [51]. In this sense, the industry leaders have adopted a new approach where they replace single power-inefficient processors with multiple efficient processors on the same chip [12]. Figure 1 [51] shows the clock rates and power consumptions of various Intel x86 microprocessor generations. *Core 2* is the first generation Intel multicore processor which is introduced after *Pentiumm 4 Presscot*.

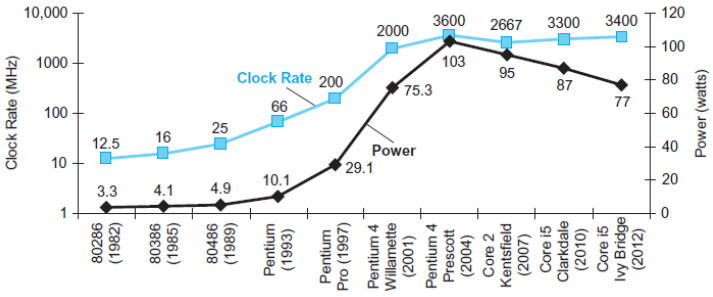

Figure 1: Clock rate and power consumption of Intel x86 microprocessors

The advent of modern processor architectures has carried the software research on parallel computing to new levels. Nowadays, modern many-core *CPU*s and parallel software that drive them have provided a significant leap forward in computing performance. Besides, *GPU*s also offer a high computing performance thanks to their architecture which is suitable for massively parallel computations.

Furthermore, as another motivation of parallel programming, the real world is massively parallel. In this sense, parallel computing fits the requirements of modelling complex real-world phenomena such as climate change, molecular dynamics and galaxy formation that include interrelated events happening at the same time with distinct temporal sequences. Parallel computing has been used to model difficult problems in science and engineering as well as industry and commercial areas. In addition to the computationally intensive problems in physics, biology and chemistry, commercial applications such as data mining, search engines, databases, financial modelling and web based business services require environments in which vast amount of data can be processed in sophisticated ways [17,

48]. Similarly, Gebali [33] states that large numbers of algorithms in the information technology area can be parallelized in order to fully benefit from new architectures and achieve higher performances.

In recent years, general-purpose distributed data processing platforms have been developed. For example, Apache Flink [50] is an open-source platform for stream and batch data processing. It is based on a streaming dataflow engine that performs core tasks such as data distribution, communication, and fault tolerance for distributed computations over data streams. Flink ecosystem includes libraries dedicated for special tasks such as graph processing, complex event processing and machine learning. Moreover, there is an experimental feature, Table API, for processing SQL-like expressions [2]. A Flink program can run on a cluster as well as a single machine and makes use of data and task level parallelisms.

While the most recent stable release of Flink, 1.0, benefits from parallel computing performed in CPUs, heterogeneous computing environments contain different processors to exploit capabilities of a range of processors such as CPUs, GPUs and *FPGA*s. In this sense, they provide an envrionment in which the workload of an application might be executed on the best fitting computing architecture. Gaster et al. [32] classify workload behaviours that an application might have into 3 groups and exemplify them as follows:

- Control-intensive: Searching, sorting and parsing
- Data-intensive: Image processing, simulation and modelling, data mining.
- Compute-intensive: Financial modelling, numerical methods

They also state that most applications contain a mix of these workloads. For example, Machine learning (*ML*) algorithms are commonly used in data mining applications. Batiz-Benet at al. [18] state that *ML* algorithms require performing computationally expensive operations. Spreading the workload over distributed systems and multiple cores becomes crucial to process large datasets. These characteristics of *ML* algorithms in the context of streaming machine learning might cause bottlenecks in the process of building or updating the *ML* model. In machine learning and data mining, the cases in which the input data changes unexpectedly in unforeseen ways are referred to *concept drift* [61]. In concept drift applications, building ML models becomes a time-critical operation since the model should not be too old and should reflect the properties of the tuples that are currently in the input stream. In some online learning scenarios, the training phase is not explicitly separated from the application phase. The *ML* model tunes itself and adapts to the new characteristics of recent tuples in the input stream when the model is being used during the application phase. In this sense, having high throughput becomes more important especially in the time-critical use cases such financial fraud detection. Concept drift applications are highly applicable in the industry. Figure 2, taken from a research paper

by Žliobaitė et al. [61], provides a taxonomy of concept drift applications based on type and industry:

Appl. / Indust.	Monitoring and control	Information management	Analytics and diagnostics
Security, Police	fraud detection, insider trading detection, adversary actions detection	next crime place prediction	crime volume prediction
Finance, Banking, Telecom, Insurance, Marketing, Retail, Advertising	monitoring & management of customer segments, bankruptcy prediction	product or service recommendation, including complimentary, user intent or information need prediction	demand prediction, response rate prediction, budget planning
Production industry	controlling output quality	–	predict bottlenecks
Education (e-Learning, e-Health), Media, Entertainment	gaming the system, drop out prediction	music, VOD, movie, news, learning object personalized search & recommendations	player-centered game design, learner-centered education

Figure 2: Categories of concept drift applications

On the other hand, there is no single computing architecture which is suitable for all different types of workloads. CPU and GPU architectures are fundamentally different from each other and therefore there is a trade-off between flexibility and computing power. Modern CPUs include many cores and perform parallel processing by implementing the Multiple Instruction Multiple Data architecture and achieve overall good levels of performance. GPUs have a Single Instruction Multiple Thread architecture in which Single Instruction Multiple Data is is combined with multithreading. GPUs offer high throughput while utilizing massive parallelism. In parallel computing, there is a close relation between underlying hardware and programming models. A high performance solution requires a complementary design considering the algorithm, computer architecture, computing model and programming model [48]. For instance, data-intensive applications can run on GPUs efficiently since their architecture design is convenient for data parallel computations in which the same program is executed in parallel on different portions of data. In this way, data access latency might be easily hidden especially in the computing intensive workloads [49].

As GPUs offer high potential to benefit data mining applications including data stream clustering, in the following subsections we first explore the use of graphics processors in general purpose applications. Afterwards, we give a conceptual overview of data stream clustering and the streaming k-means algorithm before moving on to the discussion of the motivation and contributions of our study.

3.1. Graphics Processors for General Purpose Computations

Graphics Processing Units (GPU) were initially developed for accelerating computer graphics computations while nowadays, they evolved to multi-threaded and many-core computing devices [25]. Furthermore, much work has been undertaken to utilize and improve their computing power. Using GPUs in general purpose applications is known as "General-purpose computing on graphics processing units", shortly GPGPU [59].

On the other hand, programming complexity of such platforms brings a considerable challenge for developers. Before the advent of dedicated *GPGPU* platforms, GPU computing could be performed only via 3D-rendering *API*s such as OpenGL and DirectX as long as the problem can be expressed in graphics primitives. Recently, there are platforms in which the developer can utilize the highly parallel computation power of GPUs without going through the entire graphics processing pipeline including transforming vertices, rasterization etc. [24]. Firstly, CUDA is introduced by NVIDIA in 2006. Afterwards, Khronos Gruop, which is an industry consortium including NVIDIA, introduced an open source framework called OpenCL [58].

Although these dedicated platforms reduce the programming complexity, there are still challenges which result from the architecture and the theoretical model of GPU. For example, a programmer needs to deal with challenges such as thread synchronization, manual cache usage and parallel memory access patterns. In this regard, designing efficient parallel algorithms for GPUs is a non-trivial task while technical restrictions and features of the hardware should be also taken into account in order to achieve high performance computing [48].

3.2. Data Stream Clustering

Data stream model is commonly used when modelling financial transactions, sensor networks, click streams, etc. [45, 28]. A data stream can be defined as an infinite sequence of data items. Data items are processed through operators that can be defined as continuous stream transformers in which an input stream is consumed and an output stream is produced [36].

Data stream processing differs from conventional data processing in various ways. The data arrives continuously and might evolve in time. The system must process the data within the constraints of memory and time, and it is not feasible to store all data items and evaluate them each time. Additionally, the data stream model can be used to model cases in which the entire data is available but cannot fit into the main memory [13]. There are efficient data mining techniques that are designed to overcome challenges which come

with data stream processing. For example, windowing is a concept which is commonly used for producing an approximate answer to a data stream query. In this method, a window consists of a certain number of recent data items based on count or time. In addition, windowing fits to many real world applications in terms of the semantics of the windowing mechanism. For instance, a network security application might intend to focus on more recent data in a stream [14].

Data clustering techniques allow us to understand the underlying structure of data, thus they can be used for detecting anomalies and evident features in the data. They provide a natural classification based on the degree of similarity of data items. In addition, data clustering algorithms can be used for compression purposes since they summarize the data and create cluster prototypes that can represent the data belonging to the cluster. [39]. There are thousands of clustering algorithms published in recent decades. Those algorithms can be basically divided into two groups as partitional and hierarchical. K-means is the most popular partitional data clustering algorithm which is initially proposed in 1960s by independent researchers from diverse fields, and it is still one of the most commonly used algorithms due to its simplicity, efficiency and empirical success [39]. It is commonly used in industrial practices such as pattern recognition and data analysis [15, 40]

In the recent years, advances in technology has brought high-volume and high-dimensional data in a variety of formats like text, image and video. In addition, emails, financial and commercial transactions and user-website interactions create data streams in which the nature of the data might change over time. In this respect, analysing data automatically and rapidly is a significant task. Data stream processing settings require high-speed data processing, thereby constitute another challenge for data clustering since the algorithms are expected to be computationally low-cost while the tuples are available for a limited time [34]. There are many extensions [35, 9, 22, 22] of basic algorithms such as k-means, k-medoid, fuzzy c-means, or density-based clustering, modified to cope with such challenges in data stream processing [39].

Streaming k-means algorithm is an online unsupervised learning algorithm which is an adaptation of batch k-means algorithm for streaming cases. It makes a single-pass to assign tuples to ther closest centroid, thus keeps the simplicity of batch k-means. Furthermore, streaming k-means algorithm is inherently parallel. The computational intensive part in which tuples are assigned to their closest centroid can be executed in a massively data parallel way which is efficient to execute on GPUs. In addition, some statistics that will be used for the computation of new centroids can be collected in parallel. In this way, the algorithm makes only single scan over the tuples and the cost of computing new centroids becomes dependent on the degree of parallelism. However, the related portion of the data should be transferred to GPU for each iteration when GPU is used as the com-

puting device. In streaming k-means algorithm, the machine learning model is updated by taking new data in the stream into consideration. Therefore, the clustering capacity of a system is critical feature because it affects the amount of data which the system is able to consume.

3.3. Motivation

K-means algorithm is compute-intensive and inherently parallelizable. It has been implemented on GPUs and previous research [24, 53] shows that up to 35 times speedup has been obtained as compared to corresponding CPU implementations. In addition, Hong-Tao et al. [37] propose a CUDA-based k-means implementation in which new centroids are calculated in GPU. However their approach requires extra data transfer from GPU to CPU and from CPU back to GPU for each iteration since they retrieve label array in order to reorganize tuples in CPU and transfer the tuples back to GPU to compute new centroids.

On the other hand, the streaming version of the k-means algorithm is less compute-intensive than traditional k-means algorithm. Besides, unlike the batch algorithm, it might require memory transfer between host and computing devices many times to be able to evaluate the recent data in the stream, unless a unified memory space is used.

There are two approaches used in streaming k-means implementations. Firstly, in 2009, Wu et al. [59] demonstrate, for the first time, that a GPU-based k-means implementation performs faster than CPU-based implementation in the cases where data cannot fit into GPU's onboard memory. Their approach is based on a divide and conquer methodology in which they partition the data into chunks and assign points to clusters using a GPU, and then compute centroids using host the CPU. Secondly, the implementations in which the data is partitioned and processed based on sliding windows model require a higher number of memory transfers between host and computing devices since they are likely to have more k-means iterations. The same portion of the data might be processed multiple times due to the nature of the windowing mechanism. In 2006, Cao et al. [23] propose a scalable clustering approach based on k-means using graphics processors. They also extend their work to data stream clustering and state that they observe 19 - 20 times higher performance. However, there is too little information about their streaming k-means experiment. Also, their sliding window based stream clustering experiment does not investigate the impact of streaming k-means parameters such as the number of clusters and window size on the system performance. In general, this is an early work which is done before the advent of dedicated parallel computing platforms such as CUDA and OpenCL. Furthermore, they conduct the experiments on a system with a single-core processor (Intel Pentium IV 3.4 GHz) which is far from the modern many-core CPU architectures.

Nowadays, there are heterogeneous computing devices with different characteristics. For instance, Advaced Micro Devices (AMD) offers processors in which a CPU and GPU are integrated onto the same chip. OpenCL aims a wide range of processor designs and ensures the correctness across diverse computing architectures. However, the performance of a kernel might change among different devices as OpenCL does not assure that a particular kernel always reaches its peak performance on different architectures [56].

In this thesis, we initially provide a streaming k-means implementation in OpenCL and Apache Flink which is currently not supported in the machine learning library of Flink, FlinkML [4], unlike Apache Mahout [6] and Apache Spark [3]. We implement the native algorithm in such a way that we also perform the centroid update process partially in parallel. With regard to our algorithm, we conduct preliminary experiments to obtain optimum settings for the global work size in our OpenCL application and data type in both OpenCL and Flink implementations. Afterwards, we investigate the performance on modern computing architectures by conducting a series of experiments in order to observe the impacts of the window size, the tuple size, the number of clusters and the slide size on the system throughput. We compare the performance of different computing architectures used in our OpenCL application. Besides, we compare and evaluate different systems, namely OpenCL and Apache Flink, in our streaming k-means implementation.

The main hypothesis of the thesis is that although the streaming k-means algorithm is less compute-intensive than batch k-means algorithm and requires memory transfer between host and computing device many times, we can still benefit from GPUs computational power and hide memory transfer latency. Consequently, by using GPUs, we can achieve higher throughput as compared to modern many-core CPU based OpenCL implementation as well as the Apache Flink implementation in which data and parallelisms are simultaneously exploited.

3.4. Contributions

The contributions of this thesis are the following:

1. We provide a streaming k-means algorithm that utilizes GPUs and uses sliding windows in OpenCL. Additionally, we implement streaming k-means in Apache Flink.

2. For streaming k-means algorithm, we investigate the impact of the window size, the tuple size, the number of clusters and the window slide size on throughput in many-core CPUs and modern GPUs with diverse characteristics in terms of speed and memory management. After conducting the necessary experiments using our OpenCL application, we also run the same experiments in the Apache Flink implementation to offer a comparative perspective.

3. We show that an OpenCL implementation of streaming k-means algorithm achieves higher throughputs by exploiting data parallelism than the Apache Flink implementation in which data and task parallelisms are simultaneously exploited.

3.5. Thesis Outline

Section 4. This section covers the key concepts of parallel programming and introduces Apache Flink. Next, it briefly explains the concept of heterogeneous programming and introduces OpenCL framework. Finally, it describes data stream clustering and streaming k-means algorithm.

Section 5. This section provides an overview of previous research which has been undertaken in *GPGPU* by particularly focusing on k-means clustering implementations.

Section 6. This section presents the details of the programs in which experiments are conducted. First, it gives an overview of the architecture and describes particular components. It then presents our streaming k-means algorithm implemented in OpenCL and Apache Flink, and introduces the centroid update process which is partially performed in parallel.

Section 7. This section firstly introduces the experimental setup and explains the evaluation strategy. Afterwards, it gives the results of the preliminary experiments in which we aim to obtain optimum values for parameters such as global work size with regard to our algorithm. Next, it presents the results of the experiments where we investigate the impact of the window size, the tuple size, the number of clusters and the window slide size on the system throughput. Finally, it gives an overview of the experiment results conducted in different systems and discusses the results in a comparative perspective.

Section 8. This section gives the conclusion of the thesis and suggests potential directions for future work.

4. Background

4.1. Parallel Computing

Serial or sequential computing is a model in which a problem is divided into instructions which are executed on a single processor. On the other hand, in parallel computing, a problem initially divided into discrete tasks can be executed concurrently and afterwards, each task is separated to series of instructions so that different processors execute instructions from the task part simultaneously [17]. In parallel programming, parallelism refers to a runtime property where two or more tasks can be executed in parallel, namely, at the exact same time [21].

Almasi et al., in 1988, define parallel computing as "a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (i.e., in parallel)." [11].

In addition, we can summarize common terms used in parallel computing based on "The IEEE standard dictionary of electrical and electronics" [52] and Fayez Gebali's book, "Algorithms and Parallel Computing" [33] as follows:

Parallel Software: Software that processes and transfers separate parts of a whole simultaneously using individual resources for various parts.

Parallel Architecture: Many-processor hardware that can perform parallel computing.

Thread: "A portion of a program that shares processor resources with other threads."

Process: "A program that is running on the computer."

Parallelism can be achieved using different hardware and software techniques at different levels of a computing system. Parallelism types can be classified as follows [33]:

Data-level parallelism: Data is partitioned and processed simultaneously.

Instruction-level parallelism: More than one instruction are simultaneously executed by the processor.

Thread-level parallelism: Multiple software threads are simultaneously executed by one or more processors.

Process-level parallelism: Multiple programs are simultaneously executed by one or more machines.

Parallel architectures and parallel algorithms are strictly bounded to each other. since a parallel architecture requires a parallel software that will execute it. A prominent computer architecture taxonomy was proposed by Flynn et al. [30] in 1972. This classification is based on data and instructions performed on this data. In other words, the taxonomy depends on the number of data streams and the number of instruction streams that can be simultaneously handled by the system. They propose 4 different types of architectures:

- Single instruction single data stream (SISD)
- Single instruction multiple data stream (SIMD)
- Multiple instruction single data stream (MISD)
- Multiple instruction multiple data stream (MIMD)

Processors that are based on *SIMD* architecture exploit data level parallelism. The same instructions are applied on many data chunks at the same time. On the other hand, *MIMD* architecture is more suitable for performing task level parallelism.

4.2. Apache Flink

Apache Flink [50], formerly known as Stratosphere [10], is an open source framework for parallel and distributed data processing. Flink's core is streaming dataflow engine and it has dedicated APIs for batch and stream processing. A Flink program is intrinsically parallel and distributed, and basically consists of streams and transformations. Flink can run on a cluster as well as on a single machine. The Flink system includes a *JobManager* and one or more *TaskManagers*. The textitJobManager execution of a Flink program and the *TaskManagers* executes the part of the program in parallel [1].

There are two types of local execution environments in Flink. The first one, *CollectionEnvironment*, executes Flink program on Java collections. This mode does not start full Flink runtime, thus the execution overhead is lower. However, *CollectionEnvironment* is commonly used for testing and debugging purposes since it can only work with small data which can fit into *JVM*. Besides, it uses only one thread. Secondly, *LocalEnvironment* can run a Flink program. This mode starts full Flink runtime which contains a JobManager and a TaskManager. Therefore, memory management and internal algorithms are executed in the same way that Flink runs on a cluster. *LocalEnvironment*, by default, uses as

many local threads as the number of CPU cores in the system. Besides, parallelism can be explicitly set by the user [1]. In our experiments we run the Flink engine on a single machine and use the *LocalEnvironment*.

Flink makes use of data and task level parallelisms. Flink programs are evaluated lazily, it pipelines the job. Execution strategy of a Flink program is generated by Flink's optimizer. This plan can be also visualized. Besides, it executes an internal memory management system in *JVM* [2].

4.3. Heterogeneous Computing

Heterogeneous computing refers to the systems in which conventional and specialized processors work cooperatively to achieve high computing performances. In this sense, heterogeneous computing applications can benefit from the best attributes of different types of processors [54]. The performance of a processor basically depends on the memory system and the speed of the processor. The Memory system is responsible for feeding data to the processor and is usually characterized with 2 terms, namely, *latency* and *bandwidth*. Latency refers to the delay time between the request and the access time of a memory block. Bandwidth is the maximum data transfer rate from memory to processor in a constant time[42].

Heterogeneous System Architecture (HSA) is a set of cross-vendor specifications for heterogeneous programming and aims to create an integrated programming platform in which applications can exploit parallelism while removing the CPU/GPU programmability barrier and reducing CPU/GPU data transfer latency. There are 2 kinds of compute units. Latency Compute Unit (LCU) is the generalization of CPU, and they have a large cache to reduce memory access latency. Secondly, Throughput Compute Unit (TCU) is the generalization of GPU and they are designed to maximise the data throughput while having a cost of latency. [43]

There are two common frameworks that developers can use when they write and execute programs in heterogeneous computing environments. Firstly, CUDA, which was introduced by NVIDIA in 2006, provides the developer with a more user-friendly interface for *GPGPU*. It is a general purpose parallel computing platform and programming model in which problems can be described easily and complex computations can be done efficiently [49]. However, it can run only in CUDA-enabled NVIDIA devices. Secondly, OpenCL is an open source general purpose parallel programming platform in which developers can develop portable and efficient applications that can run across various multi-core CPUs and vector processors such as GPUs. It brings substantial abstractions to reduce programming complexity in the heterogeneous environments [47]. OpenCL is developed

by Khronos Gruop which is an industry consortium including sector leaders such as AMD, NVIDIA, Apple and Intel. Fang et al. [29] give a comprehensive comparison of CUDA and OpenCL frameworks. They conclude that portability of OpenCL does not reduce its performance and it can be a good alternative to CUDA. OpenCL does not perform worse than CUDA under a fair comparison. In this thesis, we use OpenCL version 1.2.

CPU and GPU architectures are fundamentally different from each other. Modern CPUs include many cores and perform parallel processing by implementing the *MIMD* architecture. The major part of their structure consists of the control unit and cache. They achieve an overall good performance through their advance control and cache mechanisms. On the contrary, GPUs mostly consist of the arithmetic logic unit and reserve a minimal region for control and cache units. As a consequence of these differences in architecture, there is a trade-off between flexibility and computational power. CPUs are tailored for computing power while keeping general purpose functionality high and GPUs are more efficient for performing many arithmetic operations simultaneously [48].

4.4. OpenCL

The Open Computing Language (OpenCL) is a heterogeneous programming framework which is developed and currently maintained by non-profit technology consortium Khronos Group. It is cross-platform and widely supported by leaders of the industry [32].

OpenCL comes with an *API* in C language and provides many abstractions for low level hardware routines, memory management and execution models [58].Vocabulary introduced by OpenCL specifications [47] is summarized below:

Host: CPU based processor that coordinates the execution.

Device: Processors which can run OpenCL C code.

Kernel: Functions executed in the OpenCL devices.

Context: The environment which includes a set of devices, memory accessible by those devices and, execution and memory object parameters.

OpenCL specifications includes four main parts, called models, that are listed below [32, 47]:

Platform Model: Specifies the host processors and at least one device processor.

Execution Model: Defines the context and manages the execution of kernels.

Memory Model: Defines different memory types and features.

Programming Model: Defines the concurrency model and how it is mapped to hardware.

4.4.1. Platform Model

Figure 3 shows an overview of the OpenCL architecture.

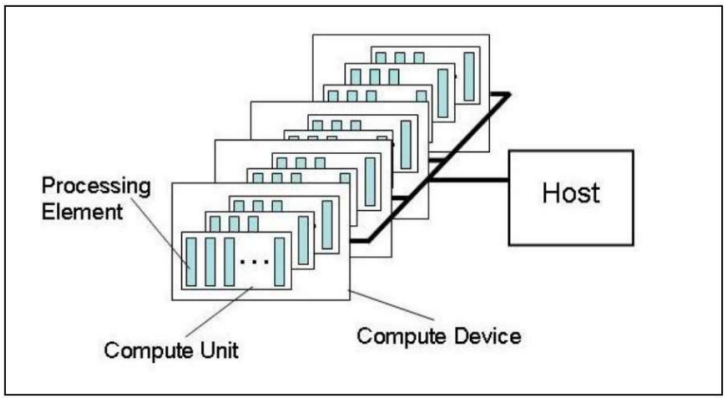

Figure 3: Platform model of OpenCL [47]

An OpenCL application consists of a host connected to one or more OpenCL devices [47]. The host application is based on CPU and controls the execution done on compute devices which can be CPU or GPU or some dedicated devices. These computing devices consist of compute units, and compute units contains one or more processing elements in which the instructions are executed.

4.4.2. Execution Model

An OpenCL program simultaneously runs on 2 different parts. One or more OpenCL devices execute the kernels while the host program runs on the host. The host application delivers the kernel to the OpenCL device and an index space, called *NDrange*, is defined. The device runs an instance of the kernel at each point of the index space. These instances are called *work item*s and they are identified by their different global IDs. Work groups consist of work items which can cooperate through shared memory and barriers if they belong to the same work group. The index space can be 1, 2 or 3 dimensional. Figure 4 explains the structure for a 2-dimensional index space [47].

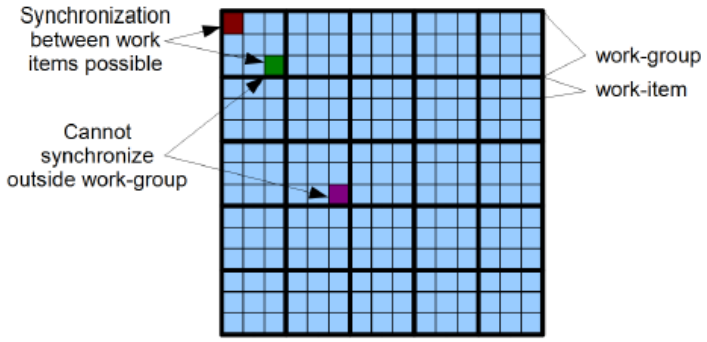

Figure 4: Structure of 2-dimensonal index space, taken from [58]

4.4.3. Memory Model

There are 4 distinct memory regions that work-items can access. The OpenCL specifications define them as follows .

Global Memory: Memory regions that can be reachable by all work items with read/write access.

Constant Memory Memory region can be reachable by all work items with read-only access.

Local Memory: Memory region can ve reachable by the work items of the work-group.

Private Memory: Memory region private to a work item.

Data must be placed one of those 4 memory regions and the location of data affects how quickly the data can be accessed. A kernel can access to the local memory faster than global memory. As long as a unified memory space is used, memory transfer between the host and the devices must be explicitly controlled. The programmer is responsible for moving data from host to device and vice versa. Mostly, the host and the computing devices have different memory spaces.

Figure 5 demonstrates the memory regions.

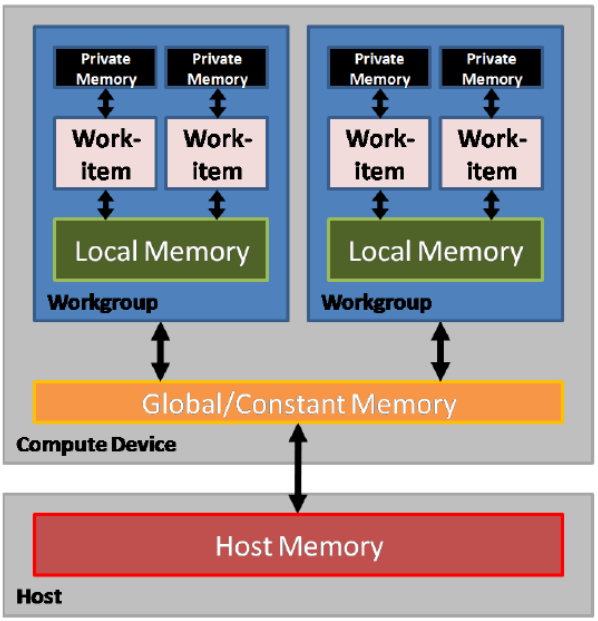

Figure 5: Memory model of OpenCL [7]

4.4.4. Programming Model

The programming model of OpenCL supports data parallel and task parallel programming models. Besides, it is possible to use a hybrid of these two approaches. Our implementation is based on the data parallel model and in this model, the execution model manages how data maps onto work items. An instance of a kernel works over the specified portion of the data. An introductory example taken from AMD OpenCL Programming Guide [7] demonstrates the conceptual difference between scalar and data parallel approaches. The scalar function given in the Listing 1 works iteratively through the *for* loop. On the other hand, the data parallel OpenCL kernel given in the Listing 2 reads only the respective elements from the input and writes the result to the corresponding part of the output. The function is called for each work item.

```
1 void square(int n, const float *a, float *result)
2 {
3   int i;
4   for (i=0; i<n; i++){
5   result[i] = a[i]*a[i];
6   }
7 }
```

Listing 1: An example of scalar fuction in C

```
kernel void dp_square
  (global const float*, global float *result)
{
  int id = get_global_id(0);
  result[id] = a[id]*a[id];
}
// dp_square exectes over "n" work-items
```

Listing 2: An example of data parallel OpenCL kernel

4.5. Data Stream Processing

Data stream is an ordered sequence of data items that can be accessed for a limited time. In the data stream model, data items arrive continuously and the system has no control over the amount and the order of the data items. The system cannot reach the data items once they are processed unless they are particularly stored in the memory. Due to the nature of the data streaming model, data stream processing differs from conventional data processing model in which data is stored and processed in the traditional database management systems which are not designed for querying fast and continuous arrival of new data elements [14]. Furthermore, data streams are infinite, and it is significant to evaluate more data items that can fit into memory. Silva et al. [55] summarize the characteristics of data streams as follows:

- The streams consists of potentially unbounded data items arrive continuously.
- The system cannot control the arrival of data items and cannot store them permanently.
- The nature of the data is likely to be nonstationary.

Considering the aforementioned challenges, Bifet et al. [20] state that data stream learning environments require working in a limited amount of time while having a limited amount of memory. The system should be ready to use the learning model any time. Babcock et al.[13, 14] explain that many applications consider very old data items as less useful and suggest 2 approaches in order to overcome restrictions in data stream processing environments. The first model is called *aging* in which data items are associated with decreasing weight constants. The second model is called *sliding window*. In this method, the last N elements in a data stream constitute a window and are processed together. They state that sliding window mechanism can semantically be part of the query. Even when this is not the case, it provides a good approximation while improving query efficiency

and reducing data volume to a reasonable size. Apache Flink DataStream API allows count and time based windowing techniques over data streams.

4.6. Data Stream Clustering and K-means Algorithm

Data clustering is a term that refers to the methods which aim to find out natural groupings of objects in a dataset. Jain et al. suggest a more statistical definition of data clustering as follows: "Given a representation of n objects, find K groups based on a measure of similarity such that the similarities between objects in the same group are high while the similarities between objects in different groups are low" [39].

K-means clustering [44] is a method in which data is automatically partitioned into k sub-groups. Jain et al. [39] introduce the k-means algorithm and summarize its steps as follows:

For a given n-dimensional object set $X = \{x\}$ and clusters set $C = \{c_{\mathrm{k}}, k = 1, ..., K\}$ that comprises K clusters, k-means algorithm discovers clusters in which the sum of the squared error between the empirical mean of a cluster and the points in the cluster is minimized. Let μ_{k} be the mean of the cluster c_{k}. The squared error between μ_{k} and the points in cluster $\mathrm{c_k}$ is as follows:

$$J(c_{\mathrm{k}}) = \sum_{x_{\mathrm{i}} \epsilon c_{\mathrm{k}}} \|x_{\mathrm{i}} - \mu_{\mathrm{k}}\|^2 \tag{1}$$

K-means algorithm aims to minimize the sum of squared error for all K clusters.

$$J(C) = \sum_{k=1}^{K} \sum_{x_{\mathrm{i}} \epsilon c_{\mathrm{k}}} \|x_{\mathrm{i}} - \mu_{\mathrm{k}}\|^2 \tag{2}$$

In this respect, the main steps of k-means algorithms are as follows:

1. Initialize centers of K clusters and repeat steps 2 and 3 until clusters do not change.
2. Assign each object to the closest cluster center.
3. Compute new cluster centers.

We can interpret the algorithm in two stages. For each iteration, firstly objects are assigned to their closest cluster center, and then, new cluster centers are computed.

There are 3 essential user-defined parameters, namely, the number of clusters, initial cluster centers, and the distance metric. Jain et al. [39] state that the basic k-means algorithm is generally used with the euclidian distance metric. The k-means algorithm is an iterative algorithm which originally ends when the clusters are stabilized. Alternatively, execution can be stopped after a certain number of iterations in order to reduce computational intensity.

Data stream clustering requires a different approach from traditional clustering. Briefly, the algorithm must take into consideration that a data stream is dynamic. Also due to the massive size of the data stream, data cannot be stored and scanned multiple times. Besides, characteristics of the data might change as time passes. Silva et al. [55] define the requirements of data stream clustering algorithms based on a number of research articles [14, 16, 57, 19] as follows:

- The algorithm should process the data items in a fast and incremental way, and provide timely results
- It should be able to detect changes in the data, namely, should discover clusters which appear or disappear.
- The algorithm should scale to the number of data items that are continuously arriving and provide a model which does not grow up by time or the number of items processed.
- It should be able to detect the outliers and handle different data types (e.g, DNA sequences, spatial and temporal data)

In general, it is possible to say that data stream clustering algorithms partially satisfy aforementioned requirements [55].

In our study, we adapted a version of streaming k-means algorithm similar to the one which is implemented by *MLlib*, Machine Learning in Apache Spark [46, 31]. We use euclidean distance (given in Appendix D) as distance metric and apply a count based *sliding windows* approach over the data stream.

We use the term tuple to refer to data items and the term centroid to refer to the center of mass of clusters. In this study, we initialize the centroids uniformly in a given interval. Our *streaming k-means* implementation then consists of 2 different phases. Firstly the tuples are assigned to their closest centroid and afterwards new centroids are computed. Finally, the centroids are updated applying a predefined rule.

Additionally, to avoid a second scan over the data, we keep simple statistics in the phase where we assign tuples to their closest centroid. We aggregate the coordinates for each dimension and also count the number elements assigned to each cluster. In this way, the computation of new centroids does not require another scan over the window and depends on tuple size, cluster count and the number of parallel instances which corresponds to parallelism in Apache Flink and global work size in our OpenCL application. Sections 6.3 and 6.4 below will explain how these parameters affects the computation of new centroids.

5. Related Work

This chapter reviews the research which has been undertaken in *GPGPU* with a particular focus on k-means clustering implementations and also briefly considers the performance of the programming languages that are used.

In 2008, Che et al. [24] provide a performance study of *GPGPU* applications in which they discuss advantages and inefficiencies of NVIDIA's computing framework CUDA and also give a comparison of k-means algorithm implemented benefiting from GPU and only using CPU. Specifically, they compare their CUDA based k-means implementation with single-threaded and four-threaded CPU versions by using a dataset from 1999 KDD Cup [5]. They demonstrate that their CUDA based GPU implementation achieves 72 times speedup as compared to single-threaded CPU implementation and 35 times speedup as compared to four threaded CPU implementation which is developed on a multicore CPU using OpenMP *API*.

In addition, Shalom et al. [53] present a research in which they exploit parallel architecture of commodity GPUs used in desktops in order to achieve an efficient k-means implementation. In this sense, they present method where they avoid the need for data transfer between CPU and GPU to compute new centroids for each iteration. They evaluate the performance using a metric which gives the computational time per iteration. For the experiments conducted using GPUs, this metric includes data transfers from CPU to GPU and vice-versa in addition to the k-means computational step where the tuples are assigned to clusters. Their implementation is based on the OpenGL API with embedded shader programs that are developed using OpenGL Shading Language (GLSL) [41]. The experiments have been conducted in an environment which consists of two systems. The first one includes 1.5 GHz Pentium IV CPU and NVIDIA GeForce FX 5900 XT GPU while the second system consists of 3 GHz Pentium IV CPU with a NVIDIA GeForce 8500 GT GPU. Therefore, they use single physical core CPUs in their experiments. They investigate the impact of the number of data points and the number of clusters and demonstrate that CPU performance is affected by the data size whereas GPU scales better to larger datasets. Precisely, in the experiments where they increase the number of points up to more than a million, they obtain up to 12 times faster performance as compared to CPU implementation using the GPU in their system with 1.5 GHz Pentium IV CPU and NVIDIA GeForce FX 5900 XT GPU. In similar settings, the GPU based implementation in the second system shows around 30 times performance gain as compared to its CPU counterpart. In addition, they conduct experiments by changing the number of cluster values between 2 and 32, and find that GPUs show almost no performance drop with the increasing number of clusters while the performance of the CPU implementations decrease as a natural consequence of the increasing number of clusters. They achieve more than 50 times performance gain when there are more than 20 clusters and finally when the

cluster count is 32, the GPU in their second system, NVIDIA GeForce 8500 GT, achieves 130 times speedup as compared to its CPU counterpart application. On the other hand, as already stated, their CPU implementations work on a single-core Intel Pentium IV processor.

In 2009, Hong-Tao et al. [37] present a k-means implementation in which they make use of the computational power of commodity GPUs using CUDA. Their approach offloads data objects assignment and centroids recalculation steps to the GPU. However, although the new centroids are calculated in GPU, the process includes an extra transfer between CPU and GPU. Their algorithm can be summarized as follows: For each iteration of k-means, after the tuples are assigned to the clusters, the label data, which shows the cluster to which each tuple is assigned to, is transferred from GPU to the host CPU. Next, the host CPU rearranges the tuples and count the total number of tuples assigned to each cluster. Afterwards both structures, namely rearranged tuples and the data which keeps the number of elements per cluster, are transferred back to GPU, so that each CUDA thread can read the data objects continuously and complete the centroid update process. They conduct the experiments on a system with 3.7 GHz Pentium D 965 which is a dual-core CPU and Geforce 8800 GTX GPU, and achieve achieve 8 to 14 times performance gain when they benefit from computational power the GPU as compared to their CPU-only implementation.

Wu et al. [59] conduct a research about accelerating the clustering of very large datasets that cannot fit into GPU's onboard memory and propose a stream-based k-means algorithm. They use a dataset which consists of one billion two-dimensional tuples and demonstrate the impact of the different parameters such as the number of clusters and the number of dimensions. Their algorithm divide data into large blocks and send it to the GPU. The GPU initially transposes the data in order to achieve a better memory access performance and then assigns each tuple to its closest centroid. Afterwards, the data which keeps the assignment of the tuples to the centroids is transferred back to the CPU which computes the new centroids. The authors conduct the experiments in two systems both with a dual quad core 2.33GHz Intel Xeon 5345 CPU and an NVIDIA GeForce GTX 280 GPU and stop the execution after 50 k-means iteration. As compared to their optimized CPU implementation which runs on 8 cores, they state that their GPU based implementation achieves 10 times speedup when the centroids fits inside GPU's constant memory and 3 times speedup when they cannot.

In 2006, Cao et al. [23] present an algorithm in which they perform distance computing and comparison in k-means algorithm using GPUs. In their algorithm, the label array, which keeps the clusters that the tuples are assigned to, is transferred to CPU and the CPU computes the new centroids for each iteration. The experiments are conducted in a system with with a 3.4 GHz Pentium IV CPU, which is a single core processor, and a NVIDIA

GeForce 6800GT GPU. The authors use synthetic datasets in which dimensionality ranges from 4 to 28, also, they conduct experiments for varying number of clusters from 8 to 256. Their GPU based implementation achieves 3 to 8 times speedup as compared to CPU based implementation. In addition, they extend their work by implementing a GPU accelerated streaming k-means algorithm which uses sliding window mechanism and state that they achieve around 20 times faster performance as compared to their CPU implementation. In this experiment they use windows which consist of one hundred thousand elements.

As shown in this review, previous work generally focuses on the batch k-means algorithm, and compares the performances of the implementations in which they use only CPU and those in which they also benefit from GPU. The proposed algorithms differ from each other in terms of the centroid update process. This step is performed in various ways which mostly involve the host CPU for different tasks. In addition, Wu et al. [59] and Shalom et al. [53] examine the impact of the parameters such as the number of tuples and the number of clusters on the performance gain. Besides, the algorithm which Wu et al. [59] use is a stream based batch k-means algorithm for the cases where the data cannot fit into GPU's onboard memory. They conduct experiments of their CPU counterpart application on 8 cores CPU and observe relatively lower speedups as compared to other studies. Finally, Cao et al. [23] look at the sliding windows based GPU accelerated streaming k-means as part of their research. However, there is not much information regarding their streaming k-means experiment. In general, these studies relied on relatively old CPUs and did not highly benefit from parallel computing which can be achieved through multicore CPUs. In our study, we particularly focus on a streaming k-means algorithm that uses sliding window mechanism and we investigate the impact of a set of parameters which consists of the window size, the tuple size, the number of clusters and the window slide size by using modern GPUs and many-core CPUs.

Regarding the programming languages, Hundt [38] presents a benchmark study which researchers at Google also contributed to. In this study, he conducts a set of experiments in which C++, Java, Scala and Go programming languages and evaluates the results in many dimensions. Finally, he concludes that C++ wins the performance comparison with a large margin especially when it is optimized. However, he states that optimizing C++ code requires a sophisticated knowledge and experience. Such impacts of the programming languages will be taken into account in discussing the results of our algorithms.

6. Implementation

This section explains the details about the system in which experiments have been conducted. It starts with an overview and introduces particular components which are the part of our experimental environment.

6.1. Architecture Overview

The system consists of four components as it can be seen in Figure 6. There are basically two main sub-systems. The first one is a streaming k-means implementation in which the algorithm is performed through instances of OpenCL kernels. The second one is a streaming k-means implementation in which the algorithm is implemented in an Apache Flink program.

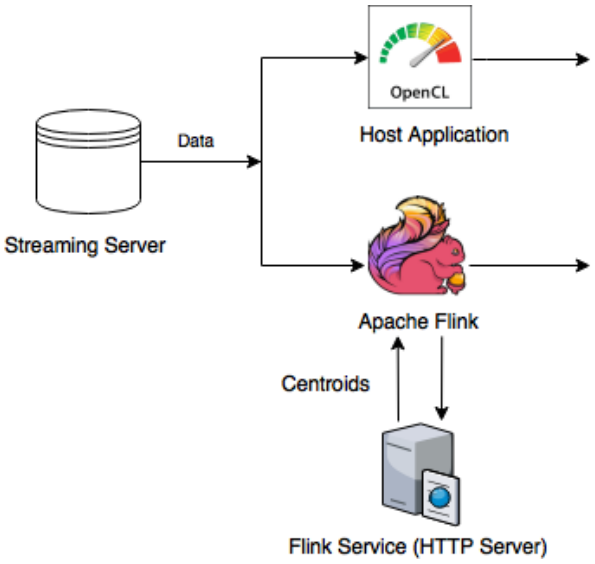

Figure 6: Overview of OpenCL and Flink streaming k-means implementations

The OpenCL implementation includes 2 components, with the Host application first receiving the data from the Streaming Server through network sockets and then running the OpenCL kernels.

Apache Flink implementation contains 3 components. The Flink program receives the data from the Streaming Server through network sockets and communicates with the Flink Service which runs an HTTP server to keep the global state of centroids.

To have a fair comparison between between our OpenCL and Flink applications, we implemented a sliding window model where the window size is strictly fixed and new tuples from the data stream are added to the portion of the tuples which are evicted as demonstrated in Figure 7.

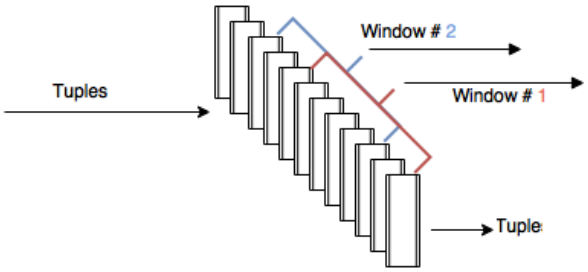

Figure 7: Sliding window concept used in the experiments

6.2. Streaming Server

Streaming server is a simple application where the data files are read from a specified directory. It writes tuple data to a network socket which uses the TCP as the communication protocol. We implement a streaming server in order to have a system that receives data through network sockets instead of reading from a file stored on local disk. This system as well as the windowing mechanism implemented in the OpenCL application can be used also for implementing different algorithms than k-means. Apache Flink offers built-in methods to read data from a socket or a local file.

6.3. OpenCL Implementation

The host application is written in C++ in which OpenCL libraries are natively supported. The overview of the workflow can be seen in Figure 8.

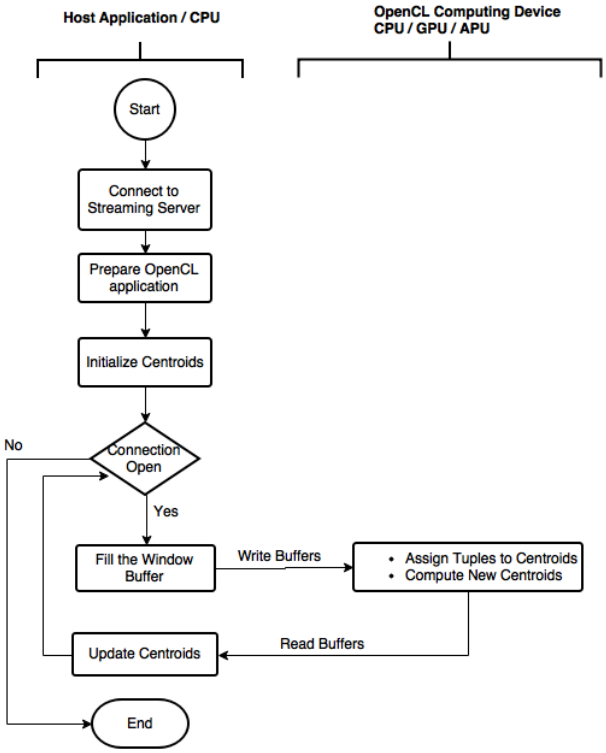

Figure 8: Overview of the OpenCL application

The host application has dedicated classes for preparing OpenCL programs and these are able to generate and run 15 different OpenCL kernels, which are tailored for particular data types to represent tuples. The application supports scalar and vector types listed below;

- Arrays of Integer Vector Types - cl_int$N$, where $N = 2, 4, 8, 16$
- Arrays of Float Vector Types - cl_float$N$, where $N = 2, 4, 8, 16$
- Arrays of Double Vector Types - cl_double$N$, where $N = 2, 4, 8, 16$
- Arrays of Scalar Types - cl_int, cl_float, cl_double

We implemented a sliding count window mechanism as presented in section 6.1 and Figure 7. The host application reads data from a TCP socket and parses the string data. Afterwards, it converts string data to integer or float. Since type cast might be a performance bottleneck for very large amounts of data, we use hand written functions based naive loops [27, 26]. Once the host application starts the execution of the OpenCL kernels, each instance accesses a particular region of the tuple input buffer and assigns tuples to their closest centroid. In addition, each kernel keeps information about the clustering locally done in the kernel instance. Afterwards, the host application computes the new

centroids using this information. This operation is independent from the number of tuples in the window and the cost depends on the global work size, the cluster count and the tuple size. How this step affects the overall performance is explained in the subsection 7.3.1.

We develop three different kernels that are used in different cases considering features such as consant memory size of a OpenCL computing device. We summarize the feature of the kernels as follows:

- *kernel-1* Uses vector structures to represent tuples and centroids and keeps centroid data in the constant memory region.
- *kernel-2* Uses array of scalar type to represent tuples and keeps the centroid data in the constant memory region
- *kernel-3* Uses array of scalars to represent tuples and centroids and keeps the centroid data in the global memory region.

We can summarize the kernel parameters and the general algorithm disregarding the structural differences in the kernels. Firstly, we look at the paramaters:

- *pointPos*: The array that keeps the tuple data.
- *KMeansCluster*: The array which keeps cluster labels that tuples are assigned.
- *centroidPos*: The array that keeps the current centroid information.
- *newCentroidPos*: The array which keeps the sum of values for each dimension, for each cluster that is generated locally by the global work item, and for each global work item.
- *tupleCountPerClus*: The array that keeps the number of elements that are assigned to clusters for each global work item.

The each kernel initially computes the indexes that will be used by the instance then follow the general algorithm which is given in the Listing 3. In addition, we give the kernel codes in Appendix A.

```
1  foreach tuple t
2    label=0;
3    dmin = FLT_MAX
4    foreach cluster c
5      d = distance(t,c)
6      if ( d < dmin )
7        update(dmin)
8        update(label)
9    end
10   newCentroid(label)+= t
11 end
```

Listing 3: General algorithm of the OpenCL kernels

As shown in the Listing 3, each kernel instance assigns tuples to their closest centroid and, for that centroid, aggregates the values of the assigned tuples for each dimension. In this way, the computation and the update process of the centroids does not require another pass over the window. The structure of the *newCentroidPos* array is demonstrated in Figure 9 where S denotes the sum of the value for the corresponding dimension.

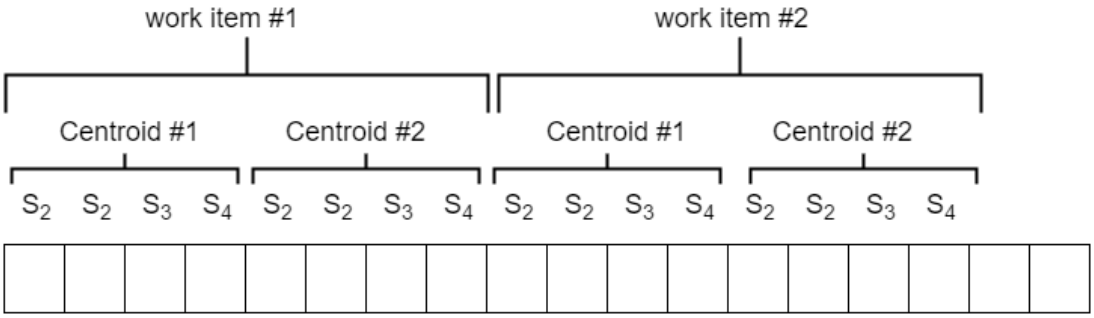

Figure 9: Structure of the array which is used for keeping aggregation values of new centroids in the OpenCL kernel

The host CPU, therefore, uses *newCentroidPos* and *tupleCountPerClus* arrays which are computed in the OpenCL kernel and updates centroids. For each centroid, it sums up the values coming from different global work items and then divides it by the total number of tuples that are assingned to the cluster. As a consequence, the cost of this operation increases with the increase in the global work size. Finally, as an update rule, it calculates the average of the old and the new centroid values.

6.4. Apache Flink Implementation

Our streaming k-means Flink implementation involves 3 different applications, namely, Streaming Server, Flink Service and Flink Program.

Current stable version oof the Flink DataStream API, 1.0, does not support sharing the global state between parallel dataflows. The parallel dataflows are run by different threads and each parallel thread communicates with an HTTP server to receive and update the global state of the centroids at the beginning and the end of each streaming k-means window. In addition, HTTP server receives the execution time logs of internal steps and logs the centroid positions.

The Flink program is based on a streaming k-means implementation developed by Andreas Salzmann and Sebastian Werner as "Big Data Analytics" class term project in Technische Universität Berlin. We extend their application to support more than 2 dimensional tuples. Furthermore, similar to our OpenCL implementation, we adopt an approach in streaming k-means steps in which we make a single pass over the tuples that exist in the window. We simultaneously assign tuples to their closest centroid and keep the simple statistics that will be used to calculate new centroids. Figure 10 gives an example plan generated by Flink Visualizer.

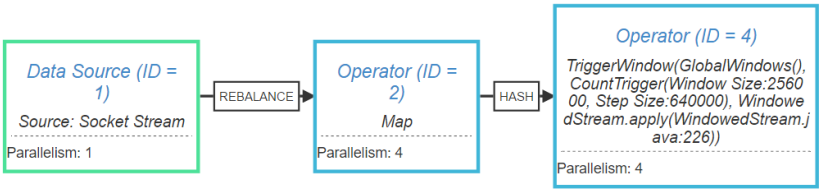

Figure 10: Execution plan of the Flink program generated by Flink visualizer

The Flink program starts with reading data from a TCP network socket. Streaming k-means Window initializes centroids in the Flink Service and applies the window function each time the window is triggered. Our implementation uses a sliding count window implementation very similar to the built-in sliding window mechanism in Flink. Once the window function is triggered, the parallel instance loads centroids from the Flink Service and assigns tuples to their closest centroid. Next, it computes new centroids locally before the Flink program sends the centroid information to the Flink Service where the centroids are globally updated and stored. Figure 11 demonstrates the window function which is the main part of the Flink application.

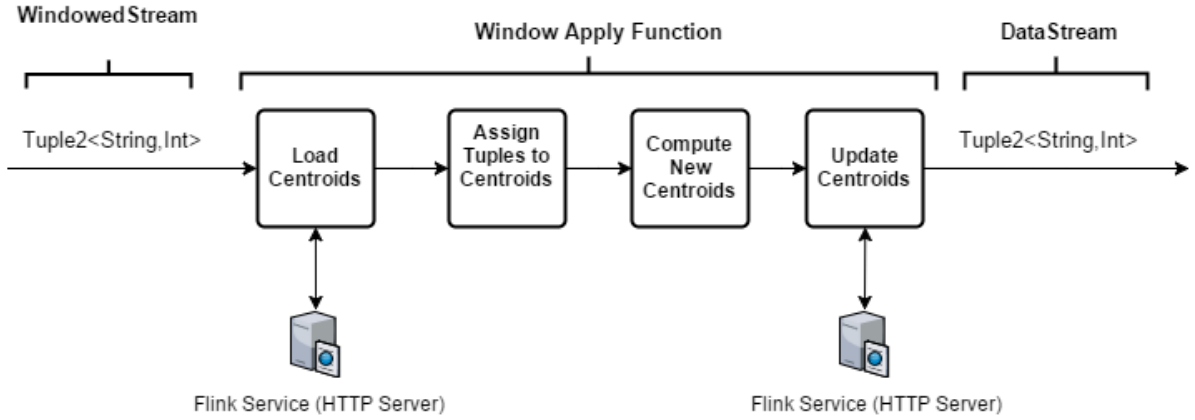

Figure 11: Workflow inside of the streaming k-means window

The input stream consists of tuples where the tuple data is represented as string and the windowing key is represented as an integer. The tuples in the output stream has the same structure. However, integer field contains the cluster number that the tuple is assigned.

Similar to the our OpenCL implementation, we sum up the values of the tuples for each dimension and for each new cluster during the phase in which we assign tuples to their closest centroid. Besides, we keep the total number of the tuples assigned to clusters. In this way, the computation of the new centroids does not require another pass over the tuples in the window. For each cluster we compute the new centroids by dividing the sum of the coordinates of the corresponding dimension by the total number of tuples that is assigned to the cluster. Finally we send new centroids to the Flink service which updates the centroids and manages the centroid broadcasting between parallel Flink instances.

6.5. Dataset Generation

We developed a Java program to generate data in which we manipulate the moving centroids and produce tuples around those based on Gaussian distribution (given in Appendix D). We also made periodic movements in the centroids again based on Gaussian distribution. The application works iteratively based on the input parameters listed below and produces the dataset as an output.

- **Step Size:** Number of the tuples that will be produced for each centroid per iteration
- **Step Count:** Number of the iterations

- **Tuple Size:** Number of elements per tuple
- **Data Type** Data type of the tuples (i.e. integer, float, double)

All iterations consist of the sub-steps described in the Listing 4. Briefly, each iteration generates the specified number of tuples for each centroid and moves the centroids.

```
foreach iteration i
  foreach step s
    foreach centroid c
      foreach dimension d
        generate c(d) + nextGaussian() * radius
      end
    end
  end

  foreach centroid c
    foreach dimension d
    c(d) = c(d) + nextGaussian() * radius
    end
  end
end
```

Listing 4: Data generator algorithm

7. Evaluation

This section will explain the evaluation setup and strategy used in the experiments before proceeding to the presentation of the results. First of all, there are 2 systems in our experiment environment. Firstly, system-1 includes an *APU* and discrete GPU. Secondly, the system-2 consists of a CPU and discrete GPU. We run the OpenCL experiments in both systems using 5 available computing devices. We run the Flink experiments in the system-2 where Flink can make use of parallelism in its many core CPU. Section 7.1 gives a broader overview about of the systems and also the parameters which have been fixed during the experiments.

In terms of strategy, we evaluate the systems in 2 phases. To begin with, we conduct preliminary experiments to obtain optimum settings and values for certain parameters such as data type and global work of the OpenCL application. This process includes 2 types of experiment.We initially investigate optimum global work sizes for different OpenCL computing devices. As explained in section 6.3, global work size affects also the centroid update phase executed in the host CPU. Besides, we evaluate overall performance of using float or integer data types in both systems, namely, our OpenCL and Apache Flink applications. Additionally for our OpenCL application, we investigate the impact of using vector structures to represent tuples and centroids. In the second phase, we examine the impact of the streaming k-means parameters on the throughput of our OpenCL and Flink implementations. Particularly, we conduct experiments with changing values of window size, tuple size, number of clusters and window slide size.

In further subsections, we present the experimental results according to the variables of interest and different systems. In particular, we refer to steps where data is read from the network and prepared to be processed in the computing device as window preparation. For instance, this includes the network cost and data type conversions. We refer to computational steps as the steps of the streaming k-means algorithm where the tuples are assigned to their closest centroid, new centroids are calculated and centroids are updated.

7.1. Experimental Setup and Datasets

During the experiments, we fix parameters such as centroid data type, window slide size and Flink parallelism. The AMD OpenCL Optimization [8] guide encourages using float types rather than double types in order to achieve higher throughput and lower memory transfer cost. We keep centroid data using float types of either array of scalars or vectors depending on the experiment setup and the tuple size. In our Flink implementation we use float data type for centroids. We do not explicitly control the local work size of

	System-1	System-2
Device Type	GPU	GPU
Device Name	AMD A10-7850K R7 GPU	GeForce GTX 980
Vendor	Advanced Micro Devices, Inc.	NVIDIA Corporation
Max Compute Units	8	16
Max Work item dimensions	256, 256, 256	1024, 1024, 64
Max Clock Frequency	720Mhz	1266Mhz
Device Type	GPU	CPU
Device Name	AMD Radeon R9 Fury	AMD Opteron(tm) Processor 6376
Vendor	Advanced Micro Devices, Inc.	Advanced Micro Devices, Inc.
Max Compute Units	56	32
Max Work item dimensions	256, 256, 256	1024, 1024, 1024
Max Clock Frequency	1000Mhz	1400Mhz
Device Type	CPU	
Device Name	AMD A10-7850 R7	
Vendor	Advanced Micro Devices, Inc.	
Max Compute Units	4	
Max Work item dimensions	1024, 1024, 1024	
Max work group size	1024	
Max Clock Frequency	3700Mhz	

Table 1: Computing devices in the System-1 and System-2

the OpenCL application and let the framework choose. Additionally, we implement the sliding window mechanism in a way that, except for the first one, each window overlaps with 75% of the previous one. In the Apache Flink implementation of our streaming k-means algorithm, we set the parallelism of the Flink job to the number of logical CPU cores in the system so that Flink can exploit data and task level parallelism for the operators in the execution plan. Besides, in the experiments conducted in Flink, we set the window size of the Flink application in a way that the total amount of the tuples that we process in the OpenCL application per window is shared between parallel Flink instances. We conduct the experiments with tuples consisting of sixteen elements. An overview of these parameters is given below:

- Centroid data type: Float
- OpenCL local work size: Chosen internally by OpenCL framework
- Window slide size: 25% of the window size (except for the slide size experiments)
- Apache Flink parallelism: The number of the logical CPU cores in the system
- Window size: 2048000 in OpenCL applications and 2048000 divided by the number of cores (parallelism) in Flink experiments (except for the window size experiments)
- Tuple Size: 16 (except for the tuple size experiments)
- Number of Clusters: 400 (except for the cluster count experiments)

Using these parameters, we run experiments on different systems and computing devices. Important features of these systems and devices are summarized in Table 1. Both systems run Ubuntu 14.04.4 LTS.

The system System-1 comprises one AMD Accelerated Processing Unit (Accelerated Processing Unit) which integrates a CPU and a GPU on a single chip. Additionally, the system has a discrete GPU, *AMD Radeon R9 Fury*, which benefits from the fast and power efficient High Bandwidth Memory (HBM) technology. The system System-2 has a many core CPU, *AMD Opteron 6376*, and a dedicated GPU, *NVIDIA GeForce GTX 980*, which is based on the Maxwell architecture.

Finally, as for the data used in our experiments, we rely on syntactical datasets generated by our application introduced in section 6.5. We run the experiments until the performance per window stabilizes. We observe that the OpenCL applications show very similar performances after the first window in which window preparation phase takes longer due to the amount of the tuples added to the window. Also, in some occasions, we observe that the kernel execution takes slightly longer for the first window and then stabilizes by the second window. In Flink experiments, we observed that the total execution time of per window usually changes in an interval which is small as compared to the average of the total execution time per window.

During the experiments, in order to make sure that data generation cost does not affect the overall performance, we occasionally send the same dataset to the application multiple times so that we can reduce the data storage cost since we do not generate data on the fly.

7.2. Evaluation Strategy

We evaluate the performance of different devices and systems in 2 phases. Firstly, we perform preliminary experiments to investigate the performance of streaming k-means implementations in OpenCL and Apache Flink with a view to determining optimum parameters which will then be taken as fixed in further experiments. Secondly, we conduct experiments for different streaming k-means parameters and examine their impact on the throughput. To do so, we define two metrics and we look at:

Computational Time per window (CPTW): Similar to the metric used in the study of Shalom et al. [53], we measure the duration of the computational steps of our streaming k-means algorithm in milliseconds. For the OpenCL application, this metric includes memory transfer costs, the kernel execution time and the centroid update step which is performed in the host CPU. For Flink experiments, this metric covers the streaming k-means steps where the tuples are assigned to their closest centroid and new local centroids are computed. Besides, it includes multiple HTTP server communication costs that occur while loading from the Flink Service and updating centroids in the Flink Service.

Throughput: Throughput gives a metric that shows the number of tuples which the system can process in one millisecond by focusing on its clustering capacity which is a crucial stage in our implementations. We formulate the throughput metric as follows:

$$\frac{\text{Window Size} \times \text{Number of Windows Processed}}{\text{Total Execution Time (ms)}} \tag{3}$$

The OpenCL application loads the computational intensive steps of the algorithm to the OpenCL computing device. The kernels assign tuples to their closest centroid and also keep the information regarding the clustering which is locally done in the kernel instance. Afterwards, the host application updates the centroids using the local statistics which come from kernel instances. This step relies on the global work size, the cluster count and the tuple size. Although the cost of this operation does not depend on the window size, the cost might become relatively high when we have high numbers of clusters, tuple size or global work size, which would affect the performance of the system. Global work size of OpenCL computation does not dramatically affect the throughput of CPU devices. On the other hand, for GPU devices, we expect to achieve a higher throughput until a certain point as long as we increase the global work size. In this sense, we initially investigate the optimum global work size that provides us with the lowest $CTPW$ in the case of our algorithm. For the Apache Flink implementation, we set parallelism to the number of logical cores in the CPU.

Secondly, we examine the performance of window preparation and computational steps in which we represent tuples using different scalar and vector data types. We only consider integer and float variations since it is not logical to perform double-precision computations while the centroids are represented as float scalar or vector data types. GPUs are tailored for floating-point computations. However, building windows where we represent tuples with different data types and structures require different operations on the host CPU. In this respect, we examine the efficiency of the system when tuples are represented with vector and array of scalars.

Based on the results of preliminary experiments as mentioned above, we fix the optimum global work sizes for OpenCL computing devices that are used in the experiments. Furthermore, we fix the data type in both OpenCL application and Flink program. Afterwards, we investigate the impact of streaming k-means parameters on the system throughput. In this respect, we firstly run experiments in which window size gradually increases and investigate the performance. The window size affects both the OpenCL computation step and the window preparation cost. Next, we examine the throughput of the system when tuples consist of various numbers of points. For this set of experiments, we run and evaluate experiments in 2 different settings since our implementation cannot benefit from the usage of constant memory space for the centroids with high dimensionality due to cer-

tain device constraints in GPUs. In the first setting, we use a kernel (described as kernel-1 in Section 6.3) in which tuples and centroids are represented using vector structure and centroids are stored in the local memory of the OpenCL device. In the second setting, we use a kernel (described as kernel-3 in Section 6.3) in which centroids are stored in the global memory of the OpenCL devices. Also, this kernel does not use vector structures, it represents tuples and centroids as array of scalars. Therefore it is more generic and provides flexibility in terms of tuple size.

Furthermore, we look at the impact of the number of clusters on the system throughput. In this sense, we aim to investigate how computational intensity affects the performance in Apache Flink application as well as in different computing devices used in our OpenCL application.

Finally, we run experiments by gradually changing the slide size. We fix the window tuple and sizes, and examine system performance while the number of tuples overlapping with the previous window changes. This parameter affects only the window preparation phase. On the other hand, it specifies how many times a tuple will belong to a window, thus, also how many times it will be processed in the streaming k-means algorithm. Evaluating a tuple multiple times, i.e. keeping its effect directly for a longer time in the centroid computation, might be beneficial depending on the application case. We investigate how this parameter affects the throughput.

7.3. Preliminary Experiments

7.3.1. Global Work Size

The number of global work sizes should be a multiple of the number of logical CPU cores (compute units) for maximum utilization [8]. In this sense, we conduct experiments with different values for each CPU device. For GPU devices, we experiment different global work size values to the obtain optimum value since it affects GPU computation as well as the centroid update which is done in the host application. We use CTPW metric in the evaluation since it covers the steps affected by global work size. We run the experiments in which windows consist of 2 million tuples with 16 elements. Cluster count is set to 400.

To begin with, Figure 12 shows the cost of centroid update done in the host CPU for increasing numbers of global work size. This process is executed in a single thread. Although the system-1 and system-2 have different CPUs, we observe that the cost of the centroid update is quite close in both systems. Figures 13 and 14 demonstrate the cost of a window in different devices.

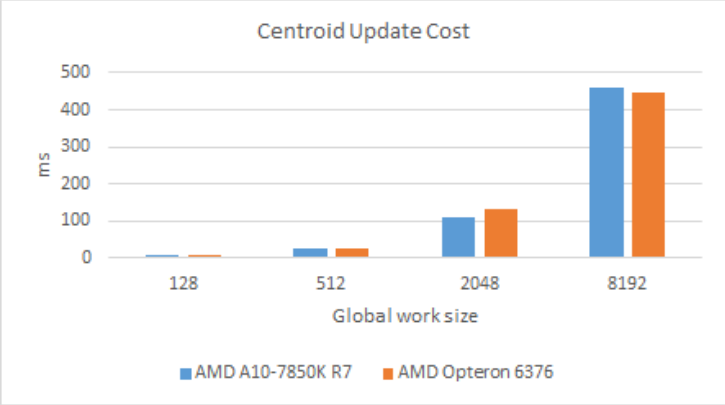

Figure 12: Cost of the centroid update in host CPU for different global work sizes

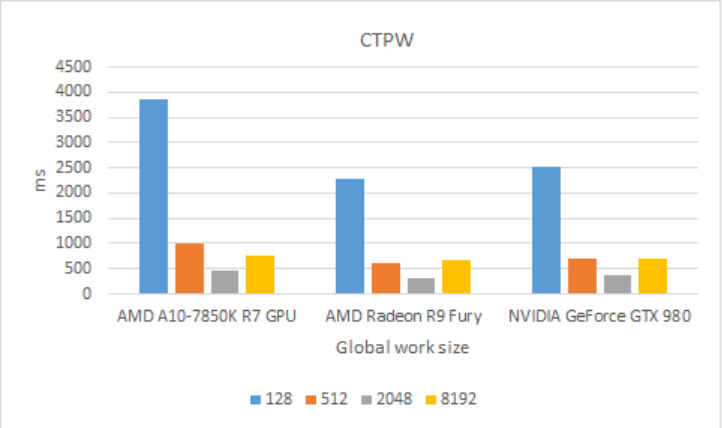

Figure 13: Computational time for one window with different global work sizes in GPUs

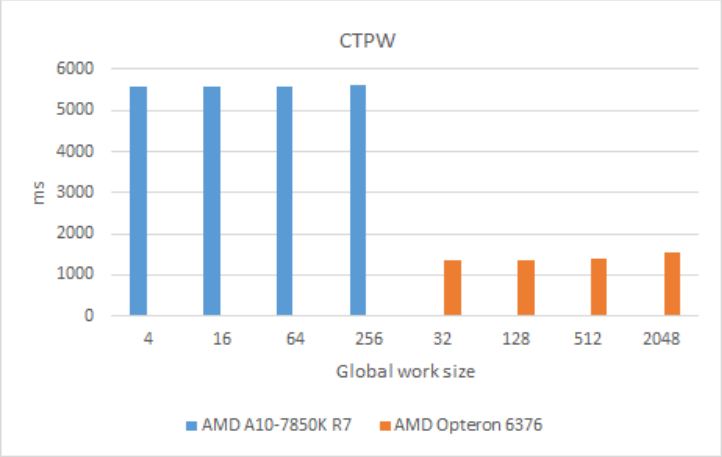

Figure 14: Computational time for one window with different global work sizes in CPUs

The experiments done in the GPU devices indicate that we can benefit from using a high number of global work size until a certain point. Afterwards the cost of the centroid update causes the computational time per window (CTPW) to increase. We observe that 2048 is an optimum value for given GPU devices in our streaming k-means algorithm. The global work size does not affect the performance of the CPU devices as it is expected and the use of a high number of global work size reduces efficiency due to the increasing cost of centroid update phase.

Consequently, we will use global work size 2048 for GPUs, 16 for *AMD A10-7850K CPU*, and 128 for *AMD Opteron 6376* CPUs in the following experiments conducted in OpenCL applications.

7.3.2. Data Type

The second set of preliminary experiments aim to investigate the effect of tuple data type in 2 dimensions. Firstly, we analyse the performance of different steps in our implementation when we represent tuples using vector types and array of scalars. Secondly we compare the cases in which integer and float data types are used to represent tuples. We equalize the network data transfer cost using the same dataset for all experiments. Also, for these experiments our variable is tuple data type, therefore centroid update cost is the same for all experiments. However, we expect to see varying CTPW measurements for different data types. Besides, we examine whether the speedup provided in the kernel execution can cover the extra cost of the window-building process executed in the host

CPU. In Flink we examine CPTW as well as the throughput metric since we do not individually measure the elapsed time in different operators of the Flink program.

We present the results for our OpenCL application in the Figure 15 for GPU devices and in the Figure 16 for CPU devices. Notations *I* and *F* refer to the cases where tuples are based on array of scalar types, integer and float respectively. Similarly, *I16* and *F16* refer to the cases where tuples are represented by corresponding vector types.

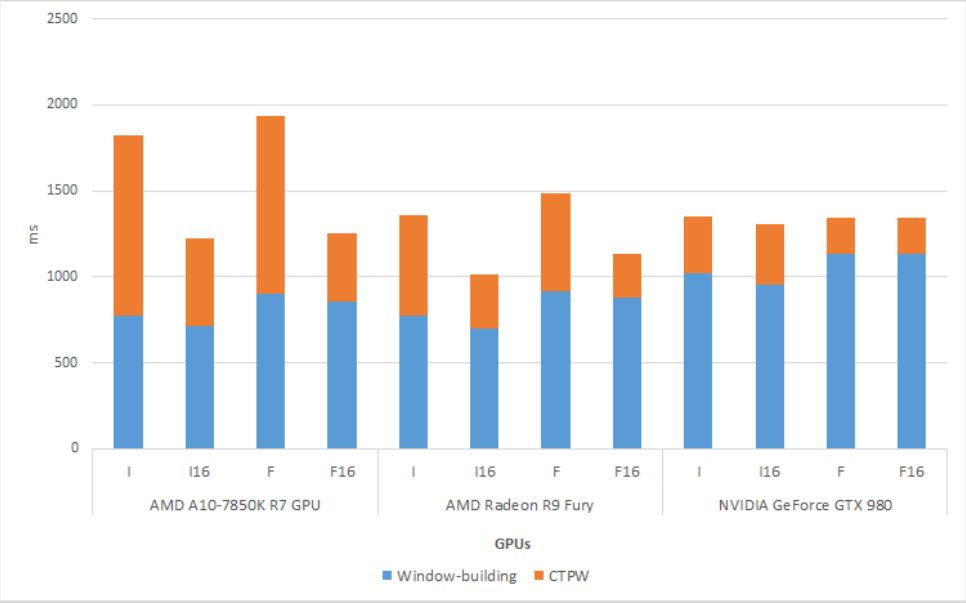

Figure 15: The impacts of the data type in GPU devices on different phases in OpenCL application

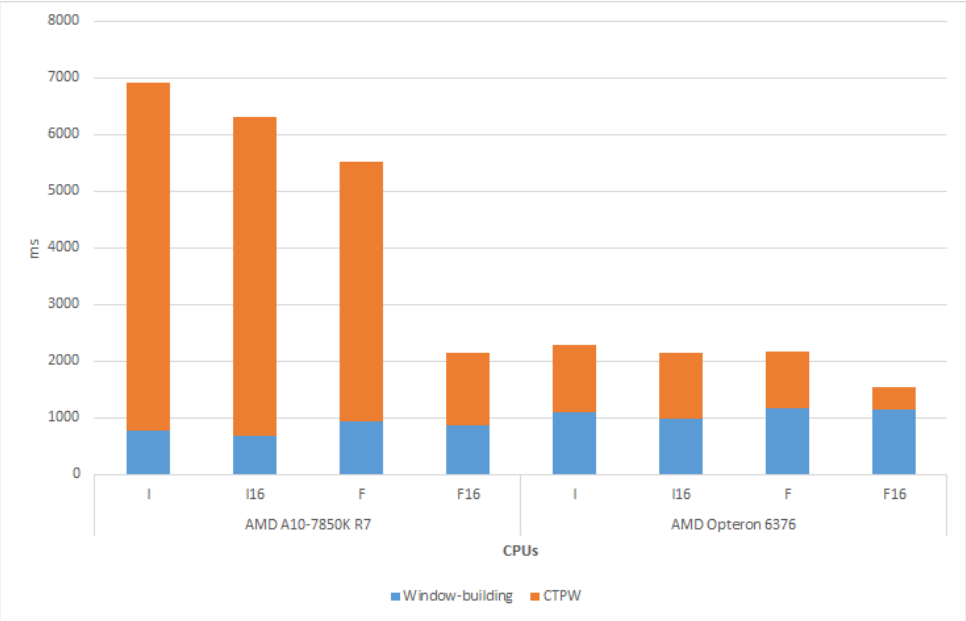

Figure 16: The impact of the data type in CPU devices on different phases in OpenCL application

Firstly, we look at window-building costs. In our implementation, this step is performed sequentially in a single thread in the host CPU. In general, we observe that for a given data type, for example integer, the cost of window-building process is slightly lower when vectors are used to represent tuples. On the other hand, using array of scalar types reduces programming complexity while providing flexibility in the number of elements in a tuple. The same pattern applies also in the float type and when we compare window-building cost of integer and float tuples, we see that float based windows are slightly more costly than integer based windows.

Secondly, we investigate the impact of the tuple data type on OpenCL computational steps in GPU and CPU devices. AMD OpenCL optimization guide [8] states that throughput and latency for 32-integer and single-precision floating operations are usually similar. Practically, float based computations are performed faster in our streaming k-means algorithm and the speedup differs from device to device. We observe that AMD devices used in the experiments make use of vectorized structures. In this sense, float vectors give the best performance in OpenCL computations done in both GPU and CPU devices. Data transfer cost does not change in any case since the size of float and integer data types provided by OpenCL API are the same.

Finally, we investigate the effect of the tuple data type in our Flink implementation. Figure 17 shows the CTPW and throughput for the cases in which windows consist of integer and float tuples. Since we do not have further information regarding the cost of the window preparation in Flink, we also consider the general system performance via throughput metric.

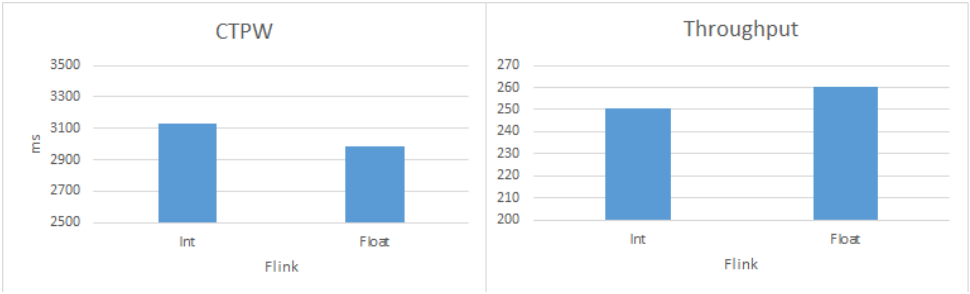

Figure 17: The impact of the data type in Flink on CPTW and throughput

We observe quite close throughput values for integer and float tuples. More specifically, the system performs slightly better with float type tuples. We believe that this might be due to the efficiency of distance calculation between float type tuples and centroids. However, further research would be required to confirm this explanation which is beyond the scope of the current study. We use float type tuples in further experiments.

7.4. Throughput Experiment Results

In this section, we give the results of the experiments in which we investigate the impact of the window size, the tuple size, the number of clusters, and the slide size. In general, we observe that Flink produces much lower throughput than OpenCL application. In this sense, we give the experiment results of Flink implementation in separated figures in order to ensure a clear visual representation.

7.4.1. Window Size

Window size, i.e., the number of tuples which a window contains, affects window preparation steps as well as the computational part which is executed in the corresponding OpenCL device. In our host application, the window preparation steps are performed in a single thread while computational steps are performed in data parallel way using diverse OpenCL devices. In this set of experiments, we fix tuple size to 16 and cluster

count to 400, and conduct experiments while the window size gradually increases. We thus examine throughputs of different systems as it can be seen in Figure 18.

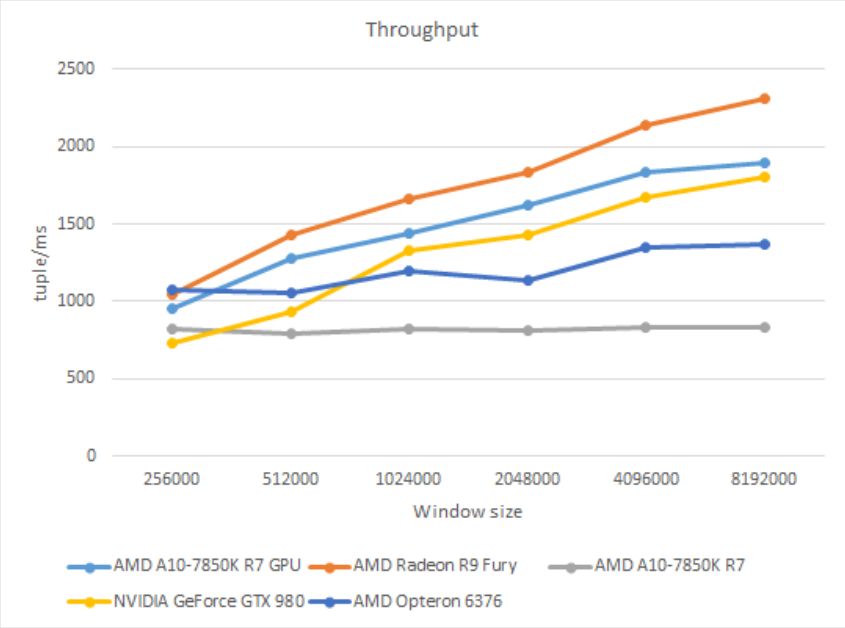

Figure 18: Impact of the window size on throughput in OpenCL application

Firstly, we observe that, for the current setup and fixed parameters, *AMD A10-7850K CPU* works in its maximum capacity. As it can be seen in Figure 16 in the section 7.3.2, *AMD A10-7850K CPU* computational steps require much more time than window preparation steps. Therefore, increasing window size does not provide any rise in the throughput since the system is already saturated with data. *AMD Opteron 6376* also shows a similar behaviour; however, its number of cores and computational power is higher than *AMD A10-7850K CPU* and its throughput slightly increases with the increase in the window size.

On the other hand, we observe that all GPU devices used in the experiments utilize their computational power better and provide higher throughputs, at least as we rise the window size up to approximately 8 million tuples. We see that CPUs may actually achieve higher throughputs than GPUs when the window size is relatively low. Especially in the system-2, *AMD Opteron 6376* shows a better performance than *NVIDIA GeForce GTX 980* which is a dedicated GPU device that is affected by memory transfer cost. In addition, *AMD Opteron 6376* benefits from vector structures as discussed in the section 7.3.2. In the system-1, GPU devices produce higher throughput than the experiments in which CPU devices are used in OpenCL applications. Besides, *AMD Radeon R9 Fury* provides

a better performance than *AMD A10-7850K R7 GPU* although *AMD A10-7850K R7 GPU* shares a unified memory space with the host CPU.

Figure 19 gives the results of window size experiments conducted in our Flink program. We observe that the window size does not dramatically affect the throughput of the system. We give the experiments results starting from windows with 1024000 tuples since, for the smaller window sizes, our HTTP server was not able to overcome the workload which is caused by simultaneously running Flink instances.

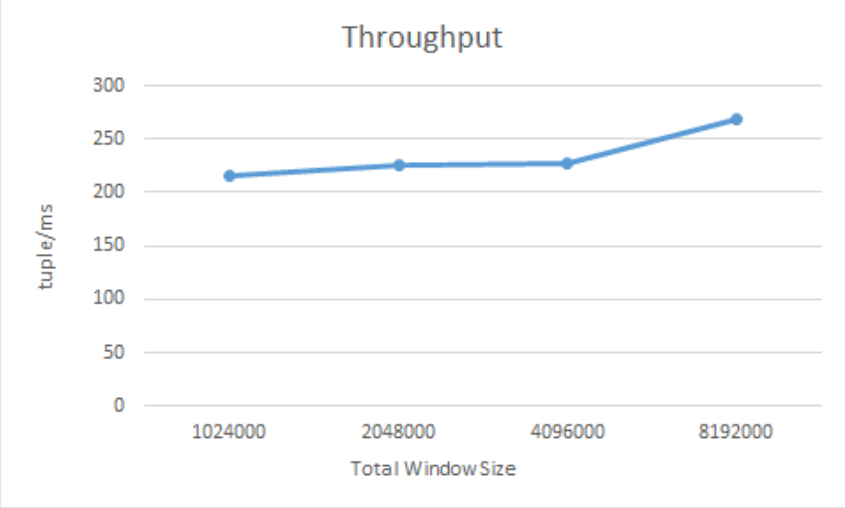

Figure 19: Impact of the window size on throughput in Flink

7.4.2. Tuple Size

In the experiments in which we observe the impact of the tuple size on the system performance, we use 2 different kernels depending on the number points that a tuple contains. Up to sixteen points, we use vector structures to represent tuples and centroids to benefit from vendor and device specific optimizations. Also, we keep centroid data in the local memory of the OpenCL device to provide faster memory access. However, as this region is limited, it might not be well suited to hold centroid data when the tuple size and the number of clusters are high. Our second type kernel is thus more generic. It represents tuples and centroids as array of scalars, uses only global memory of the OpenCL device and works with all tuple sizes. In both experiment settings, we set the cluster count to 400. Figure 20 and Figure 21 show the experiment results for different tuple sizes.

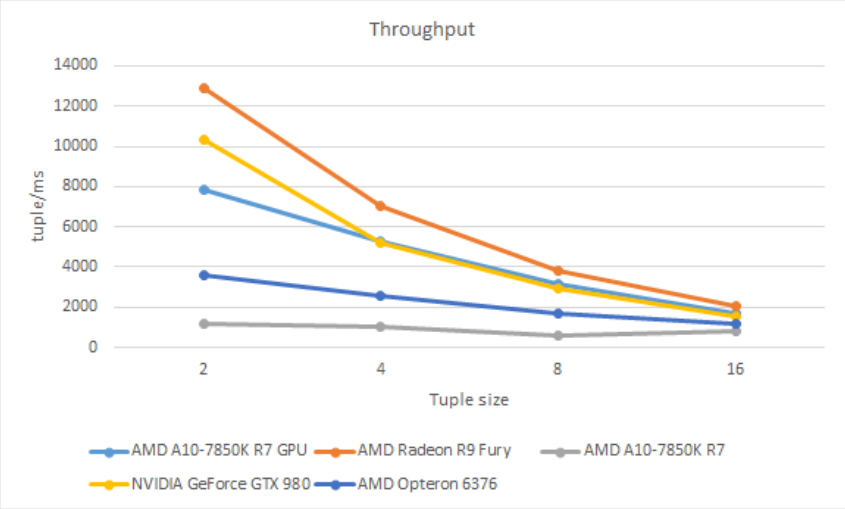

Figure 20: Impact of tuple size on throughput, in OpenCL application (lower tuple sizes)

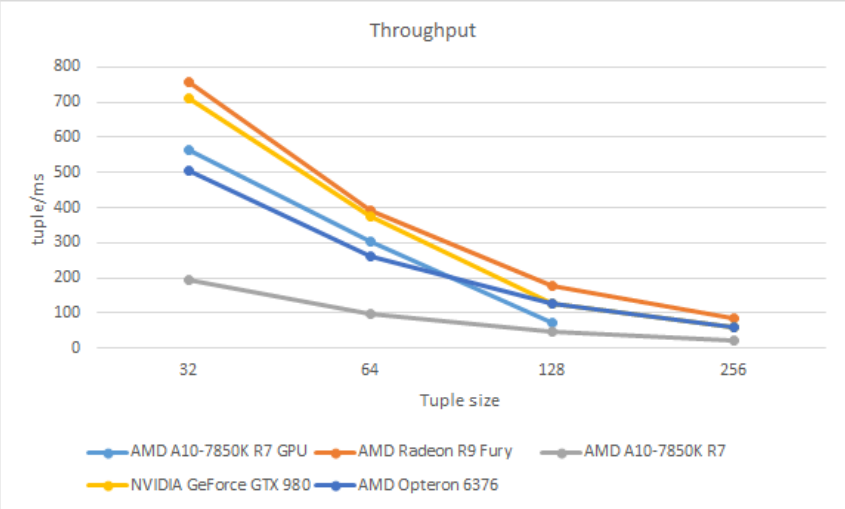

Figure 21: Impact of tuple size on throughput, in OpenCL application (higher tuple sizes)

Figure 20 shows the throughput results where the tuples consist of 2,4,8 and 16 elements. GPUs produce high throughputs for smaller tuple sizes and for the relatively larger tuple sizes, they still perform much faster than CPUs. However, the gap between the performance of GPUs and CPUs shrinks when the tuple size grows. Yet, the lowest GPU throughout remains 40% higher than the highest CPU performance. Data transfer cost for dedicated GPUs rises with the increase in the tuple size. We observe that the throughput produced by CPUs shows a slower decrease as compared to that produced by GPUs.

Figure 21 demonstrates the results for larger tuple sizes in which we leave aforementioned kernel optimizations for GPUs aside. As expected, the proportional throughput difference between GPU and CPU devices decreases. Large tuples sizes increase the intensity of the operations performed in a loop which is not encouraged for GPUs. Similarly to the case for small tuple sizes, increasing tuple size causes the cost of data transfer to rise, thus, the cost of the centroid update phase which is perfomed in host CPU becomes more apparent. We observe that in the system-1, the CPU, *AMD Opteron 6376*, produces even slightly higher throughput than *NVIDIA GeForce GTX 980*. Also, GPUs lose their efficiency faster than CPUs in case when the tuples contain more elements. Figure 21 does not include the results of the experiment conducted in *AMD A10-7850K R7 GPU* since the experiment did not run successfully due to the memory constraints of the device.

Our Flink implementation represents tuples and centroids as array of scalars in the computation. The throughputs of the system for different tuple sizes can be seen below in Figure 22.

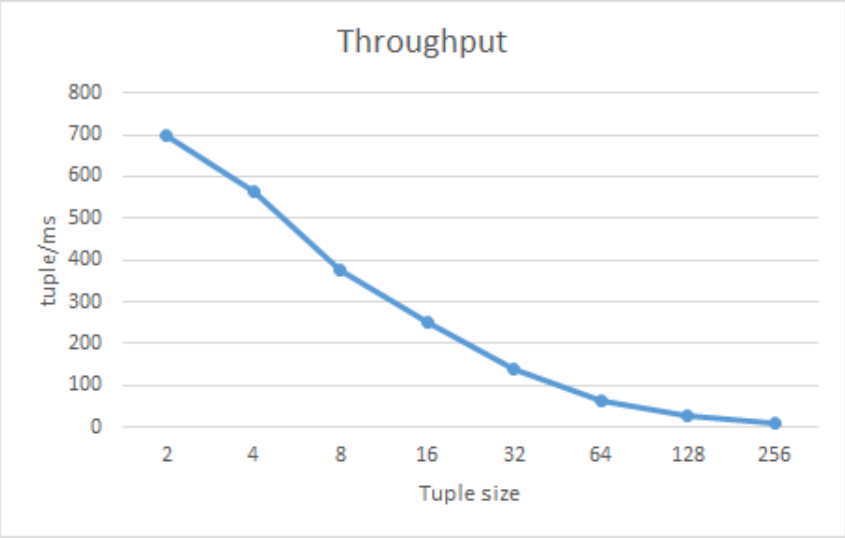

Figure 22: Impact of the tuple size on throughput in Apache Flink

Our Flink implementation shows a sharper decrease in throughput for relatively small tuple sizes with the trend of decrease becoming flatter for higher tuple sizes. Besides, increasing the window preparation overhead in the Flink engine, we observe that the network communication cost between parallel flink instances and the Flink Server, HTTP server that keeps the global state of the centroids, becomes more evident due to the increasing workload in the Flink Service side. However, this step is still far from being a bottleneck as compared to the streaming k-means computational step, and computational

steps account for the major part of the total execution time. The details regarding the internal steps are given in Appendix C.

7.4.3. Number of Clusters

In these experiments, we observe how computational intensity due to the cluster count affects the system throughput. Figure 23 shows the results for the OpenCL implementation.

Figure 23: Impact of the tuple size on throughput in OpenCL application

In our OpenCL streaming k-means implementation, *AMD Radeon R9 Fury* produces the highest throughput for all conducted experiment cases. *AMD A10-7850K R7 GPU* performs a faster clustering than *NVIDIA GeForce GTX 980* especially in the lower number of clusters. On the other hand, we observe that CPUs give relatively closer throughputs to the *NVIDIA GeForce GTX 980* when the number of clusters is relatively low. For example, *AMD A10-7850K CPU* can achieve a higher throughput than the discrete GPU which exists in the system-2 when the number of clusters is set to 100. However, it should be noted that those devices belong to the different systems in which window preparation is performed through different processors. An overview regarding the efficiency of the window preparation phase in the system-1 and system-2 is given in Appendix B.

Figure 24 shows the results of cluster count experiments done in our Flink program, in which we observe a linear decrease in throughput as the number of clusters increases.

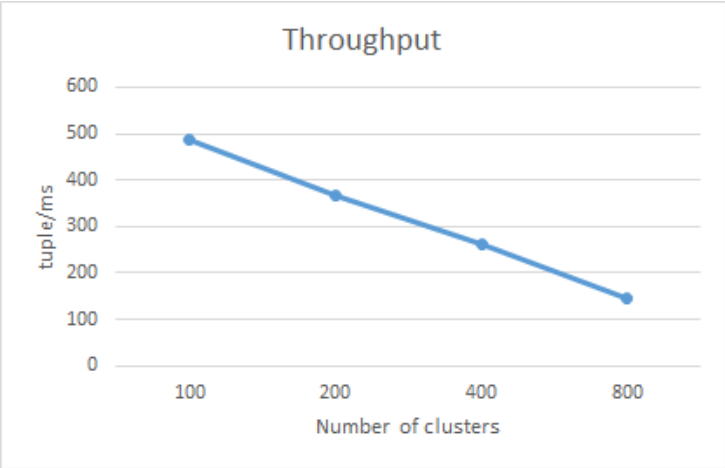

Figure 24: Impact of cluster count on throughput in Apache Flink

7.4.4. Window Slide Size

Step size refers to the number of tuples that will be processed for the first time in a window. In this set of experiments, we fix the tuple size and type, the cluster count and the window size. In this regard, slide size parameter mainly affects the total window preparation while the cost of computational steps of the algorithm stays constant. For small slide sizes, the cost of window preparation is lower since only a minor portion of the window is updated. Therefore, a tuple is evaluated many times. Depending on the application, this might be intended to increase the quality of the output. We conduct experiments with slide size parameters which correspond to 1/64, 1/32, 1/16, 1/8, 1/4, 1/2 of the window size.

We evaluate these experiment results in 2 dimensions. Figure 25 shows throughput of the systems with various slide sizes. Additionally, Figure 26 demonstrates a modified version of the throughput metric, called throughput-2, for this particular experiment case. Precisely, it shows the number of tuples that are evaluated for the first time in a window per millisecond. In this sense, we aim to analyse the trade-off between throughput and the handling of fast streaming data sources. In sliding windows mechanism, a tuple is inherently evaluated multiple times, therefore this parameter determines the lifetime of a tuple.

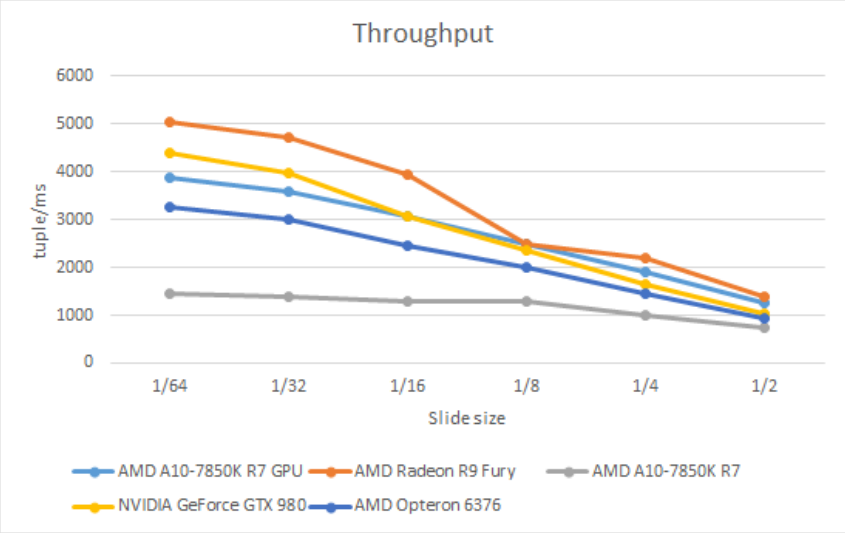

Figure 25: Impact of the slide size on throughput in OpenCL application

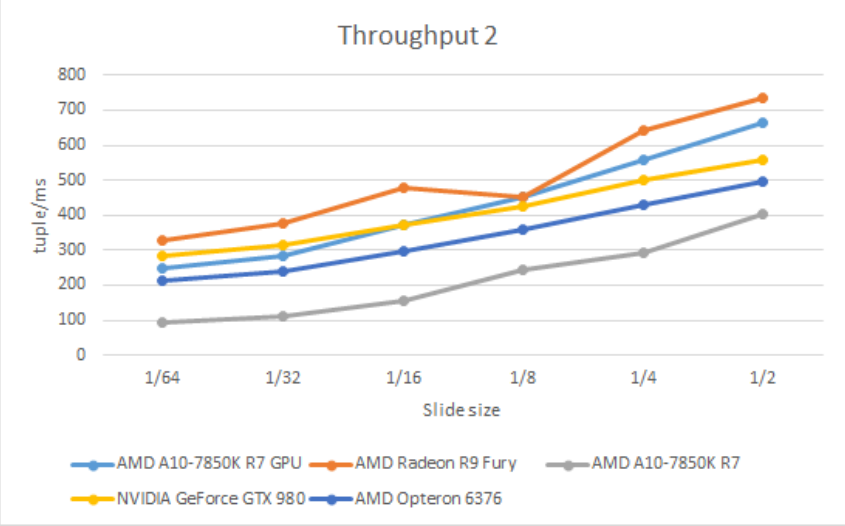

Figure 26: Impact of the slide size on throughput-2 in OpenCL application

Naturally, Figure 25 demonstrates that any increase in the slide size decreases the throughput. Besides, computational workload per window is not affected by the changing slide size parameter. We observe that the same pattern applies to all particular devices used for the OpenCL application. However, for the given experiment settings, window preparation is far from becoming a bottleneck. As we see in Figure 26, the capacity of a system to evaluate a tuple in a window for the first time increases with the rise in the slide size.

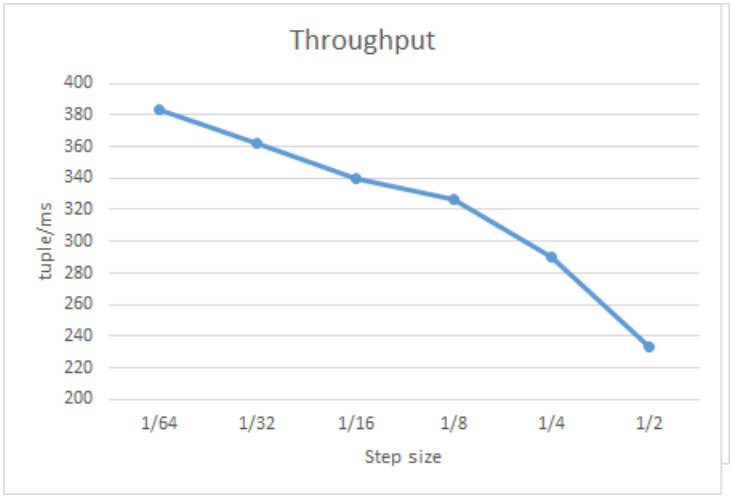

Figure 27: Impact of the slide size on throughput in Flink

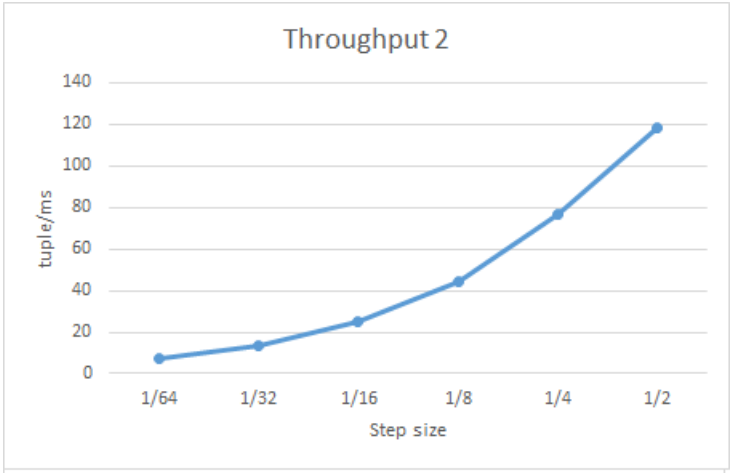

Figure 28: Impact of the slide size on throughput-2 in Flink

Flink application shows a similar behaviour with throughput decreasing and Throughput 2 increasing as the slide size increases. Precisely, we observe a sharper decrease in the throughput after step the size goes beyond the 1/8 of the window size in our experimental settings. This shows that the efficiency of the window preparation phase lags behind the computational phase. However, the high number of new elements in each window is still enough the provide an increase in the throughput-2 metric.

7.5. OpenCL and Flink Performance Comparison

In this subsection, we firstly cover the algorithmic differences between our OpenCL and Flink implementations. Afterwards, we give a performance comparison of OpenCL applications that use *AMD Radeon R9 Fury* and *AMD Opteron 6376*. We then demonstrate the speedups achieved in our OpenCL applications as compared to the Flink implementation, and finally discuss the impact of different parameters on the throughput with regard our OpenCL experiments.

In terms of the algorithmic features of OpenCL and Flink implementations, the main difference takes place in the centroid update process. In the OpenCL application, this step is performed only once for each window, whereas in the Flink implementation, Flink Service keeps the global state of the centroids and performs the update process. Flink Service is an HTTP server that communicates with individual parallel Flink instances, therefore its workload is affected by the parallelism differently from the OpenCL application. In this sense, centroid update is performed for each parallel Flink instance which processes windows, and this situation brings an extra overhead to the Flink implementation. On the other hand, as an advantage, window preparation is performed simultaneously in many threads in Flink while this is performed sequentially in a single thread in the OpenCL applications.

In addition to the aforementioned differences, Apache Flink is a general-purpose batch and stream data processing engine which is written in Java. It is designed to support a wide range of applications thanks to its broad support to operators as well as data types and structures. On the other hand, our OpenCL application is written in C++ and designed for such a streaming application as the one we perform. Comparing the performance of two different programming languages requires a comprehensive research which covers many aspects. In general, C++ in shown to perform much faster than several programming languages including Java especially when the program is optimized, as discussed in Section 5.

Before comparing the OpenCL and Flink implementations, it would also be useful to give a performance comparison of OpenCL applications that use *AMD Radeon R9 Fury*, which is the GPU device that produces the highest throughput in our experimental setup, and those which use *AMD Opteron 6376* which is the CPU device that produces higher throughputs than the other CPU. Table 2 gives an overview of the speedups that *AMD Radeon R9 Fury* achieves in different experiments.

As discussed in section 7.4, for our OpenCL application, CPUs may achieve higher throughput then GPUs devices in some occasions while overall, GPUs performs faster. Meanwhile, for our streaming k-means implementation, OpenCL applications that use

Parameter	Value at the lowest throughput	Lowest speedup	Value at the highest throughput	Highest speedup
Window size	256000	0.97	8192000	1.68
Tuple size (1-16)	16	1.71	2	3.58
Tuple size (32-256)	256	1.36	32	1.50
Number of clusters	100	1.50	200	1.59
Step size (Throughput)	1/8	1.24	1/16	1.16
Step size (Throughput-2)	1/8	1.26	1/16	1.59

Table 2: Comparison of the OpenCL applications, *AMD Radeon R9 Fury* vs *AMD Opteron 6376*

CPU as the computing device can show competitive performance with the cases in which GPU devices are used as the computing device in the application. However, it should be noted here that *AMD Radeon R9 Fury* and *AMD Opteron 6376* exist in the different systems in which window preparation is performed through different processors. An overview regarding the efficiency of the window preparation phase in the system-1 and system-2 is given in Appendix B.

Regarding the comparison between the OpenCL and Flink applications, the Figures that show the speedups in corresponding experiments are given below.

Firts, Figure 29 shows the speedup that OpenCL applications achieve in the experiments that we run with different window sizes.

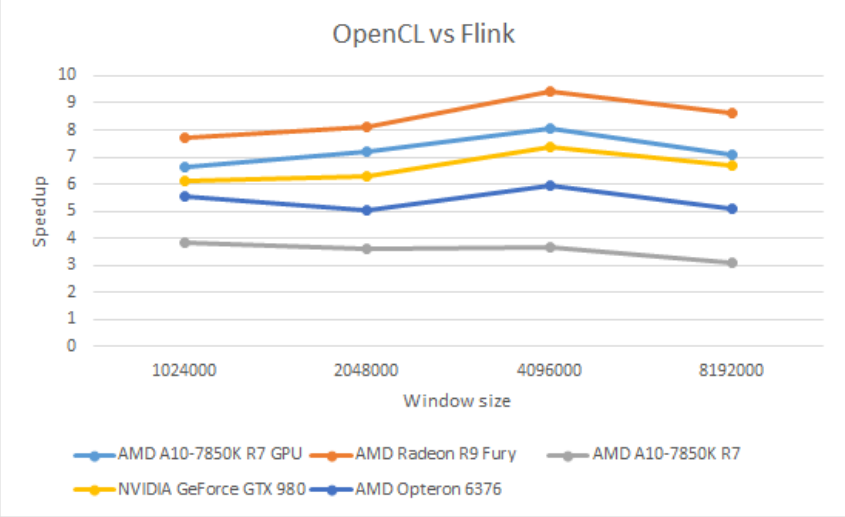

Figure 29: OpenCL vs Flink throughput comparison for window size experiments

Figure 30 and 31 display the speedup achieved in the tuple size experiments for the OpenCL application where different computing devices are used.

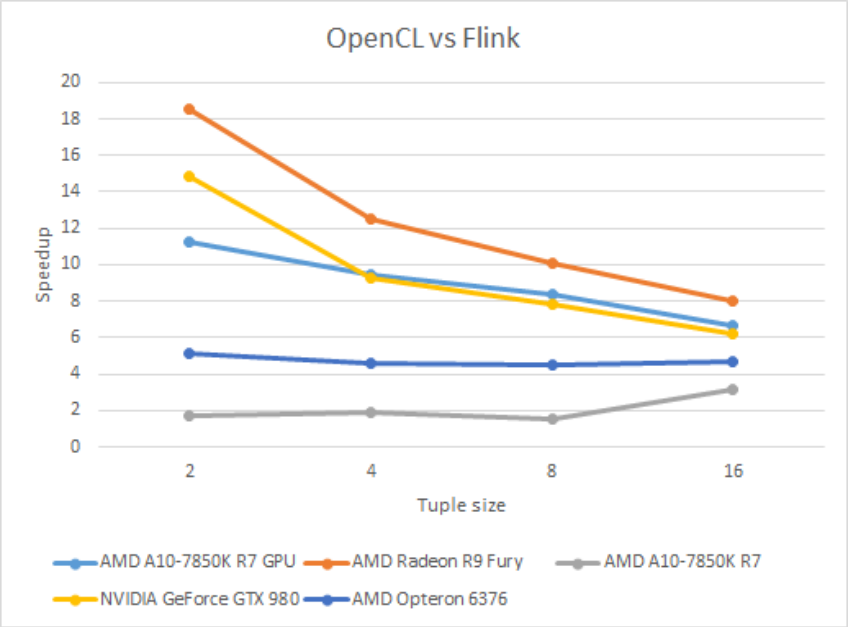

Figure 30: OpenCL vs Flink throughput comparison for tuple size experiments (lower tuple sizes)

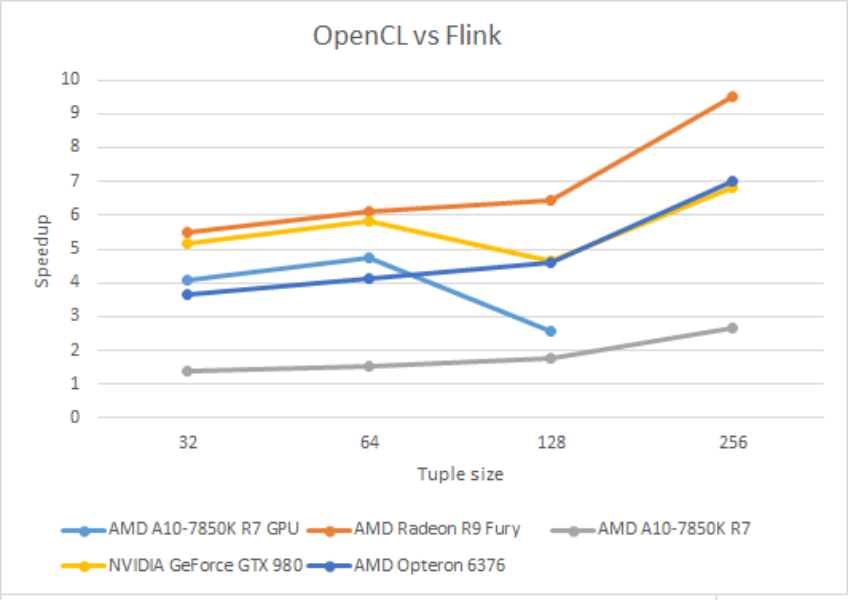

Figure 31: OpenCL vs Flink throughput comparison for tuple size experiments (higher tuple sizes)

Figure 30 shows the speedups for smaller tuple sizes in which a more optimized kernel is used in the computations. We observe that, GPUs provide a higher speedup especially in small tuple sizes. On the other hand, OpenCL applications where CPUs are also used as the computing device show a more stable speedup pattern. Figure 31 shows that although the kernel used for the tuples sizes larger than 16 does not benefit from certain optimizations, we observe that the speedup achieved in the OpenCL applications increases for most devices. Especially for the tuples with more than 128 elements, the rapidly increasing workload of the Flink Service is also arguably influential in increasing the OpenCL speedup.

Figure 32 shows the speedup achieved in the OpenCL application in the experiments conducted with increasing numbers of clusters.

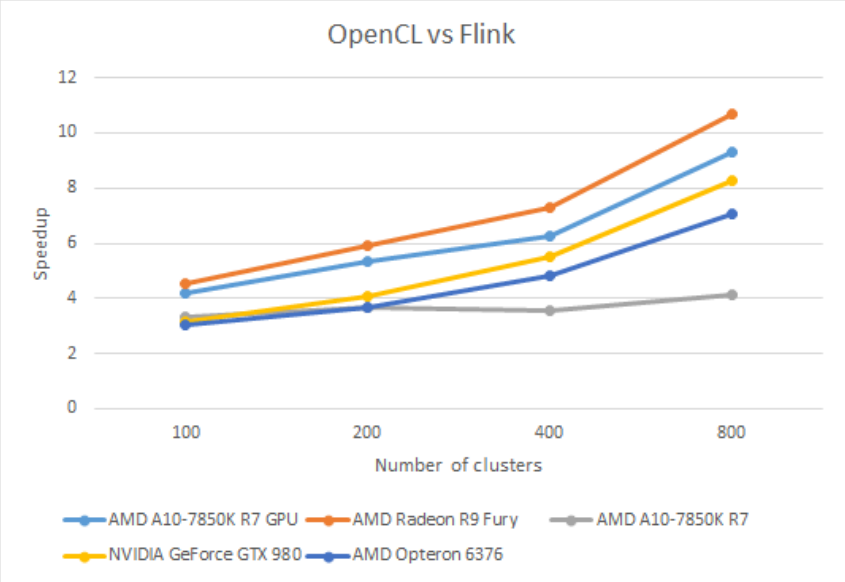

Figure 32: OpenCL vs Flink throughput comparison for the number of clusters experiments.

The number of clusters does not dramatically affect the performance of the Flink Service since the tuple size is set to 16 in the experiments. More specifically, the step where tuples are assigned takes, on average, 30 times longer than the total Flink Service communication cost when cluster count is 800. However, it mainly affects the streaming k-means computation where tuples are assigned to their closest centroid. The speedup achieved in our OpenCL applications, except for the one where *AMD A10-7850K CPU* is used, becomes higher as the number of clusters increases. Details regarding the elapsed time in the internap steps are given in Appendix C.

Finally, Figure 33 and Figure 34 present the throughput comparison for window slide size experiments considering both the throughput metric and the throughput-2 metric which is calculated for this specific set of experiments.

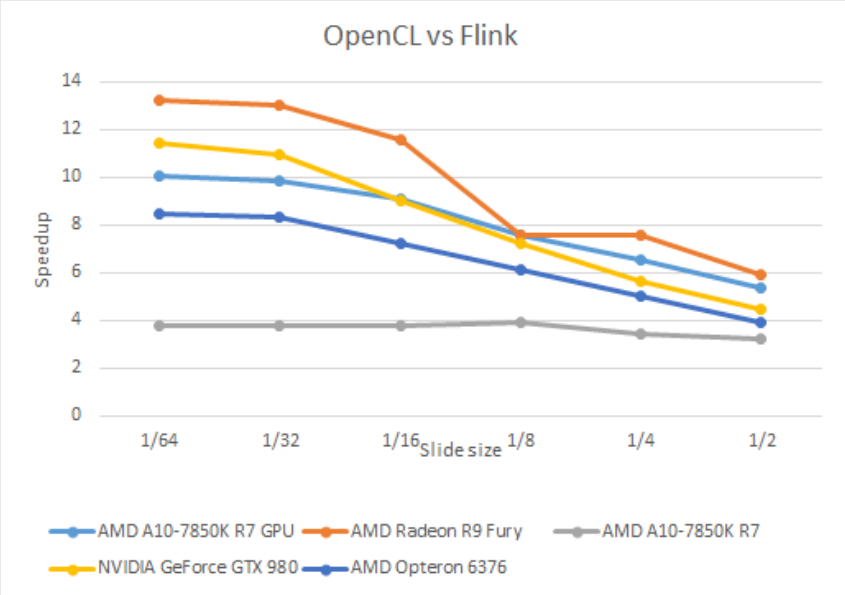

Figure 33: OpenCL vs Flink throughput comparison for slide size experiments

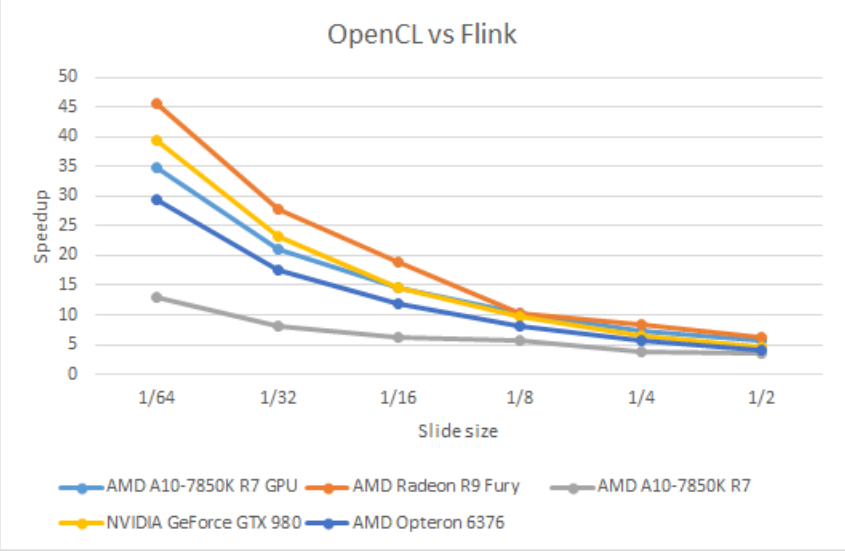

Figure 34: OpenCL vs Flink throughput-2 comparison for slide size experiments

Parameter	Value at the lowest throughput	Lowest speedup	Value at the highest throughput	Highest speedup
Window size	1024000	7.6	4096000	9.4
Tuple size (1-16)	16	8.0	2	18.5
Tuple size (32-256)	32	5.5	256	9.5
Number of clusters	100	4.5	800	10.6
Step size (Throughput)	1/2	5.8	1/64	13.1
Step size (Throughput-2)	1/2	6.1	1/64	45.5

Table 3: Flink vs OpenCL (GPU, *AMD Radeon R9 Fury)*

Parameter	Value at the lowest throughput	Lowest speedup	Value at the highest throughput	Highest speedup
Window size	2048000	5.0	4096000	5.9
Tuple size (1-16)	8	4.4	2	5.1
Tuple size (32-256)	32	3.6	256	6.9
Number of clusters	100	3.0	800	7
Step size (Throughput)	1/2	3.9	1/64	8.4
Step size (Throughput-2)	1/2	4.1	1/64	29.2

Table 4: Flink vs OpenCL (CPU, *AMD Opteron 6376*)

We compare the performance of the OpenCL and Flink implementations based on the experiments in which we change the window slide size. This only has an impact on window preparation steps where the cost of the computational steps stay constant. Flink performs windows preparation in parallel while in the OpenCL application this process is executed sequentially using a single thread. In this comparison, we observe the that speedup achieved in the OpenCL applications is reduced with the increase in the slide size.

Table 3 gives an overview of the performance comparison of Flink and the OpenCL application that uses *AMD Radeon R9 Fury*, which is the device that produces the highest throughputs in our experimental setup. Similarly, Table 4 gives an overview of the performance comparison of Flink and the OpenCL application that uses *AMD Opteron 6376* which is a many-core CPU in which also Flink experiments have been conducted.

The parameters that have been tested in the throughput experiments affect the workload of the algorithm in different ways. Before we conclude, we summarize the impact of our parameters on throughput with regard to our OpenCL application as follows:

Window size: We show that GPUs scale better to larger windows. Their throughput increases with increases in the window size. When we compare CTPW for CPUs and GPUs, we observe that CPUs display a nearly linear increase while computational power of GPUs can handle the increasing window size better and provides very close kernel execution times until a certain point before starting to increase. Appendix B.1 gives the experiment results with individual costs of the window preparation and computational phases.

Tuple size: We conduct the tuple size experiments with 2 different kernels as discussed in section 7.4.2. In general the throughput decreases with increases in the tuple size. For smaller tuple sizes, we observe that the window preparation is mostly the bottleneck for GPUs, whereas for CPUs, it is the computational step which causes the bottleneck. For larger tuple sizes, a similar pattern applies for both CPUs and GPUs. However, the computational efficiency of the GPUs decreases due to the increasing intensity of loop operations as well as the changing memory region in which the centroids are stored. On the other hand, the efficiency of CPUs is not necessary affected in the same way. The loop operations can even have a positive impact when CPUs are used in the kernel execution. Details regarding the internal steps of this experiment are given in Appendix B.2.

Number of clusters: In the experiments we conduct by changing the number of clusters, the cost of the window preparation is constant. The throughput of the system decreases in all experiments as the number of clusters increases. However, this decrease is smaller than the decrease caused by increases in the parameter of number of tuples due to the constant window preparation cost. Appendix B.3 gives the result regarding the elapsed time in window preparation and computational phases.

Window slide size: The slide size does not affect the kernel execution time per window but increases the window preparation cost. Also, it determines how many times a tuple will be evaluated in a window and keeps its effect directly in the model. As we already discuss in Section 7.4.4, the increasing slide size causes the throughput to decrease. However, the capacity of the system to evaluate a tuple for the first time increases since new tuples will be added to the window faster.

8. Conclusion

The use of graphics processing units, GPUs, in general purpose applications provides substantial performance enhancement in scientific computing as well as data mining applications such as clustering. Although the GPU computing brings certain drawbacks such as memory access latency and programming complexity, the undertaken research shows that the k-means algorithm, which is one of the most commonly used algorithms due to its simplicity, efficiency and empirical success, can benefit from massively parallel computation ability of GPUs.

Streaming adaptation of k-means algorithm requires memory transfer between host and computing devices many times to be able to evaluate the recent data in the stream, unless a unified memory space is used. The native algorithm consists of two phases where the tuples are assigned to centroids and the new centroids are computed respectively. The first phase can be inherently expressed in a data parallel way. In this thesis, we provide streaming k-means implementations which use sliding window mechanism in OpenCL and Apache Flink. We present an algorithm in which centroid computation is also performed partially in parallel. We give comprehensive performance study which covers a range of relevant parameters. More specifically, we look at the impact of window size, tuple size, number of clusters and window slide size in our OpenCL implementation, which is experimented using different computing devices, and the Apache Flink, implementation which runs on a modern many-core CPU and benefits from data and task level parallelism. We evaluate the results in two dimensions. We firstly compare the results of the OpenCL application in which different CPUs and GPUs are used as the computing device. Afterwards, we also compare the throughputs of our OpenCL and Apache Flink implementations.

Overall, we achieve higher throughputs in our OpenCL application when we use GPUs as the computing device. However, *AMD Opteron 6376*, which is a 32 core CPU, can occasionally produce higher throughouts than GPUs especially when the window size and the number of clusters are relatively low. In general, as different from the previous studies presented in Section 5, we observe that the speedup that is achieved when GPUs are used as the computing device is reduced as compared to the counterpart CPU implementation that runs on modern CPUs. Precisely, the OpenCL application that uses *AMD Radeon R9 Fury* produces 0.97 to 3.58 times of the throughput that is produced by the OpenCL application which uses *AMD Opteron 6376* in similar settings.

Secondly, we compare the performance of our OpenCL implementations in which we use *AMD Radeon R9 Fury* and *AMD Opteron 6376* with the Apache Flink implementation. Considering the throughput metric, we observe that our GPU based OpenCL application achieves 5 to 18 times speedup as compared to the Flink application. Also, our CPU based OpenCL application achieves 3 to 8 times speedup as compared to the Flink application.

We realise that the HTTP Server in the Apache Flink that keeps the global state of the centroids requires further optimizations to be able to overcome the workload which occurs in certain conditions when too many HTTP requests come from parallel Flink instances in a short time. In addition, the systems currently are not protected against the value overflow that might happen during centroid update operations.

Furthermore, the window preparation part in our OpenCL implementation is currently executed sequentially in a single thread, but this processes can be implemented in parallel in order to improve the performance of the window preparation phase, therefore increase the overall throughput of the system. Moreover, OpenCL kernels require specific optimizations to achieve their peak performance in computing devices with different features. Future work could thus focus on device and vendor specific optimizations on the OpenCL program preparation and computation phases in order to obtain maximum gain in the each device in terms of the individual internal steps of the execution such as memory access and kernel execution. Such studies would make it possible to understand the dynamics of system performance in various computing architectures from different perspectives.

References

[1] Apache flink: Scalable batch and stream data processing. Available at: http://flink.apache.org/.

[2] Basic api concepts. Available at: https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/common/index.html Accessed: 2016-06-18.

[3] Clustering - spark.mllib. Available at: http://spark.apache.org/docs/latest/mllib-clustering.html Accessed: 2016-07-23.

[4] Flinkml - machine learning for flink. Available at: https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/batch/libs/ml/index.html Accessed: 2016-07-23.

[5] Kdd cup 1999 data. Available at: http://kdd.ics.uci.edu/databases/kddcup99/ Accessed: 2016-07-28.

[6] Streamingkmeans algorithm.
Available at:https://mahout.apache.org/users/clustering/streaming-k-means.html Accessed: 2016-07-23,.

[7] Introduction to opencl programming guide, 2010. Available at:http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/opencl-course-introduction-to-opencl-programming/ Accessed: 2016-03-05.

[8] Amd opencl programming optimization guide, Aug 2015. Available at: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide Accessed: 2016-07-07.

[9] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 81–92. VLDB Endowment, 2003.

[10] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[11] George S Almasi and Allan Gottlieb. Highly parallel computing. 1988.

[12] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[13] Brain Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 234–243. ACM, 2003.

[14] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[15] Piyush P Baramkar and DB Kulkarni. Review for k-means on graphics processing units (gpu). In *International Journal of Engineering Research and Technology*, volume 3. ESRSA Publications, 2014.

[16] Daniel Barbará. Requirements for clustering data streams. *ACM sIGKDD Explorations Newsletter*, 3(2):23–27, 2002.

[17] Blaise Barney. Introduction to parallel computing. Available at: https://computing.llnl.gov/tutorials/parallel_comp/ Accessed: 2016-07-18.

[18] Juan Batiz-Benet, Quinn Slack, Matt Sparks, and Ali Yahya. Parallelizing machine learning algorithms.

[19] Albert Bifet. Adaptive stream mining: Pattern learning and mining from evolving data streams. In *Proceedings of the 2010 conference on adaptive stream mining: Pattern learning and mining from evolving data streams*, pages 1–212. Ios Press, 2010.

[20] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 11:1601–1604, 2010.

[21] Clay Breshears. *The art of concurrency: A thread monkey's guide to writing parallel applications.* " O'Reilly Media, Inc.", 2009.

[22] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, volume 6, pages 328–339. SIAM, 2006.

[23] Feng Cao, Anthony KH Tung, and Aoying Zhou. Scalable clustering using graphics processors. In *International Conference on Web-Age Information Management*, pages 372–384. Springer, 2006.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[25] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[26] Tino Didriksen. C convert string to double speed. Available at: https://tinodidriksen.com/2011/05/28/cpp-convert-string-to-double-speed Accessed: 2016-06-16.

[27] Tino Didriksen. C convert string to int speed. Available at: http://tinodidriksen.com/2010/02/16/cpp-convert-string-to-int-speed/ Accessed: 2016-06-16.

[28] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. An auto-tuning solution to data streams clustering in opencl. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 587–594. IEEE, 2011.

[29] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

[30] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[31] Jeremy Freeman. Introducing streaming k-means in apache spark 1.2, 2015.

[32] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[33] Fayez Gebali. *Algorithms and parallel computing*, volume 84. John Wiley & Sons, 2011.

[34] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):515–528, 2003.

[35] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 512–521. IEEE, 1999.

[36] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[37] Bai Hong-Tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He. K-means on commodity gpus with cuda. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 3, pages 651–655. IEEE, 2009.

[38] Robert Hundt. Loop recognition in c++/java/go/scala. 2011.

[39] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[40] Liheng Jian, Cheng Wang, Ying Liu, Shenshen Liang, Weidong Yi, and Yong Shi. Parallel data mining techniques on graphics processing unit with compute unified device architecture (cuda). *The Journal of Supercomputing*, 64(3):942–967, 2013.

[41] John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language. *Language version*, 1, 2004.

[42] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, CA, 1994.

[43] George Kyriazis. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, 2012.

[44] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[45] Alireza Rezaei Mahdiraji. Clustering data stream: A survey of algorithms. *International Journal of Knowledge-based and Intelligent Engineering Systems*, 13(2):39–44, 2009.

[46] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[47] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[48] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(02):285–329, 2014.

[49] CUDA Nvidia. Programming guide, 2008.

[50] Seif Haridi Asterios Katsifodimos Volker Markl Kostas Tzoumas Paris Carbone, Stephan Ewen. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, "38"("4"):28–38, 2015".

[51] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[52] Jane Radatz. *The IEEE standard dictionary of electrical and electronics terms*. IEEE Standards Office, 1997.

[53] SA Arul Shalom, Manoranjan Dash, and Minh Tue. Efficient k-means clustering using accelerated graphics processors. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 166–175. Springer, 2008.

[54] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal*, 2006(142):7, 2006.

[55] Jonathan A Silva, Elaine R Faria, Rodrigo C Barros, Eduardo R Hruschka, André CPLF de Carvalho, and João Gama. Data stream clustering: A survey. *ACM Computing Surveys (CSUR)*, 46(1):13, 2013.

[56] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[57] Dimitris K Tasoulis, Niall M Adams, and David J Hand. Unsupervised clustering in streaming data. In *null*, pages 638–642. IEEE, 2006.

[58] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 2012.

[59] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering billions of data points using gpus. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6. ACM, 2009.

[60] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005.

[61] Indrė Žliobaitė, Mykola Pechenizkiy, and João Gama. An overview of concept drift applications. In *Big Data Analysis: New Algorithms for a New Society*, pages 91–114. Springer, 2016.

Appendix

A. OpenCL Kernels

This section gives source code template that is used during the program execution to generate desired OpenCL kernel. Since they are generated dynamically, the program replaces some special string keys according to parameters such as window size, global work size and tuple size. Descriptions of these string keys are given below. Listings 5 and 6 give source code templates corresponding to kernel-1 and kernel-2.

- *VECTYPE_POINT*: Data type of the points (e.g. float, integer, integer8, float16)
- *VECTYPE_CENTROID*: Data type of the centroids (e.g float, float2, float4)
- *TYPE_DIST*: Data type of the distance metric (e.g float)
- *block*: The number of tuples that should be processed by per global work item. This value is calculated via equations below respectively for kernel-1, and kernel-2 and kernel-3

$$block = \frac{windowSize}{global\_work\_size[0]} \tag{4}$$

$$block = \frac{tupleSize \times windowSize}{global\_work\_size[0]} \tag{5}$$

- *cCount*: The number of clusters
- *tupSize*: The number of the elements each tuple contains

```
__kernel
void assignCentroid(
    __global VECTYPE_POINT *pointPos,
    __global unsigned int *KMeansCluster,
    __constant VECTYPE_CENTROID *centroidPos,
  __global VECTYPE_CENTROID *newCentroidPos,
  __global unsigned int *tupleCountPerClus
    )
{
    unsigned int gid = get_global_id(0) * block;
  unsigned int gidNewCentroid = get_global_id(0) * cCount;
  for (int j = 0 ; j < block ; j++ ){
    // Load 1 point
    VECTYPE_POINT vPoint = pointPos[gid+j];
```

```
    TYPE_DIST leastDist = FLT_MAX;
    uint closestCentroid = 0;

    for(int i=0; i < cCount ; i++)
    {
      VECTYPE_CENTROID tempCent = centroidPos[i];
      TYPE_DIST dist =(vPoint.s0 - tempCent.s0) * (vPoint.s0 - tempCent.s0) +
                (vPoint.s1 - tempCent.s1) * (vPoint.s1 - tempCent.s1) +
                (vPoint.s2 - tempCent.s2) * (vPoint.s2 - tempCent.s2) +
                (vPoint.s3 - tempCent.s3) * (vPoint.s3 - tempCent.s3) ;
      leastDist = fmin( leastDist, dist );
      closestCentroid = (leastDist == dist) ? i : closestCentroid;
     }

    KMeansCluster[gid+j] = closestCentroid;
    newCentroidPos[gidNewCentroid+closestCentroid].s0 += vPoint.s0 ;
    newCentroidPos[gidNewCentroid+closestCentroid].s1 += vPoint.s1 ;
    newCentroidPos[gidNewCentroid+closestCentroid].s2 += vPoint.s2 ;
    newCentroidPos[gidNewCentroid+closestCentroid].s3 += vPoint.s3 ;
    tupleCountPerClus[gidNewCentroid + closestCentroid]++;
  }
}
```

Listing 5: OpenCL kernel-1 (four-elements vector type)

```
__kernel
void assignCentroid(
    __global VECTYPE_POINT *pointPos,
    __global unsigned int *KMeansCluster,
    __constant VECTYPE_CENTROID *centroidPos,
  __global VECTYPE_CENTROID *newCentroidPos,
  __global unsigned int *tupleCountPerClus
    )
{
    unsigned int gid = get_global_id(0) * block;
  unsigned int tupleStartIndex = gid / tupSize;
  unsigned int gidNewCentroid = get_global_id(0) * cCount * tupSize;
  unsigned int gidNewCentroid2 = get_global_id(0) * cCount ;

  for (int j = 0 ; j < block / tupSize ; j++ ){

    VECTYPE_POINT tempPoint[tupSize];
    for ( int n =0; n < tupSize ; n++ ) {
        tempPoint[n] = pointPos[gid+ j * tupSize + n];
    }

    TYPE_DIST leastDist = FLT_MAX;
      uint closestCentroid = 0;

```

```
25      for(int i=0; i < cCount ; i++)
26      {
27         TYPE_DIST dist = 0;
28       for ( int n =0; n < tupSize ; n++) {
29         TYPE_DIST tempDist = tempPoint[n] - centroidPos[i*tupSize+n];
30         dist +=  tempDist * tempDist;
31       }
32       leastDist = fmin( leastDist, dist );
33       closestCentroid = (leastDist == dist) ? i : closestCentroid;
34      }
35
36     KMeansCluster[tupleStartIndex + j] = closestCentroid;
37     tupleCountPerClus[gidNewCentroid2 + closestCentroid]++;
38
39     for ( int n =0; n < tupSize ; n++) {
40     newCentroidPos[gidNewCentroid + closestCentroid * tupSize + n] += tempPoint[n];
41       }
42
43   }
44 }
```

Listing 6: OpenCL kernel-2

B. Additional Results for OpenCL Experiments

Figure [?] gives the window preparation cost of the CPUs in system-1 and system-2. It the average cost of the 16 windows which includes the first window that is naturally more costly than the others due to our sliding window implementation. The window size is set to 2048000 and the slide size is set its 25% as they are the fixed parameters in our experiments.

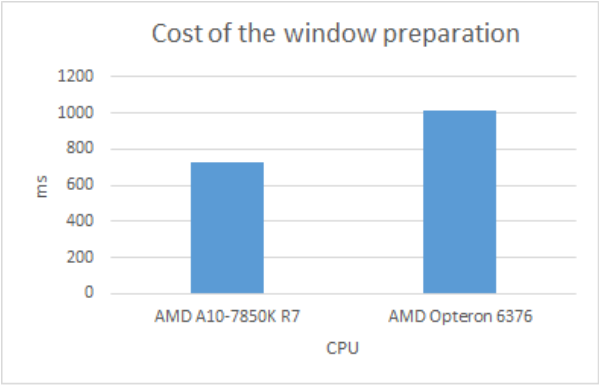

Figure 35: Window preparation cost of different CPUs in OpenCL application

B.1. Cost of the Internal Steps in the Window Size Experiments

The figures below give the average elapsed times for window preparation and computational phases in the window size experiments conducted in our OpenCL application.

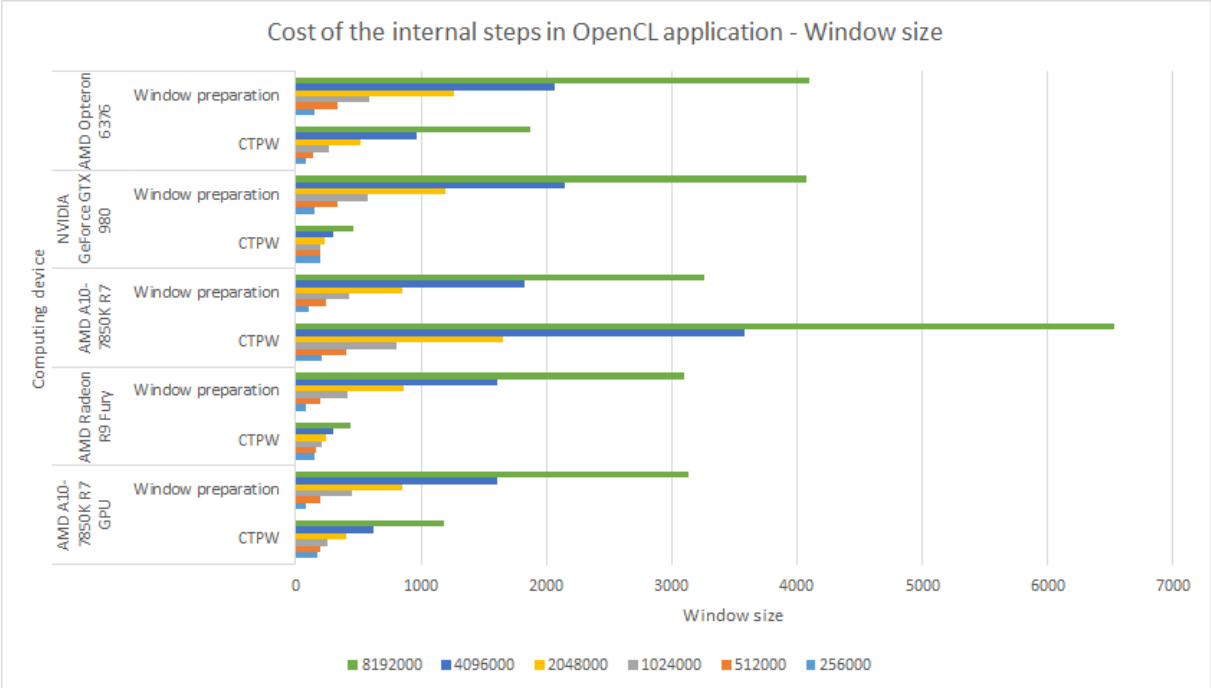

Figure 36: Cost of the internal steps in OpenCL application, window size experiments

B.2. Cost of the Internal Steps in the Tuple Size Experiments

The figures below give the average elapsed times for window preparation and computational phases in the tuple size experiments conducted in our OpenCL application.

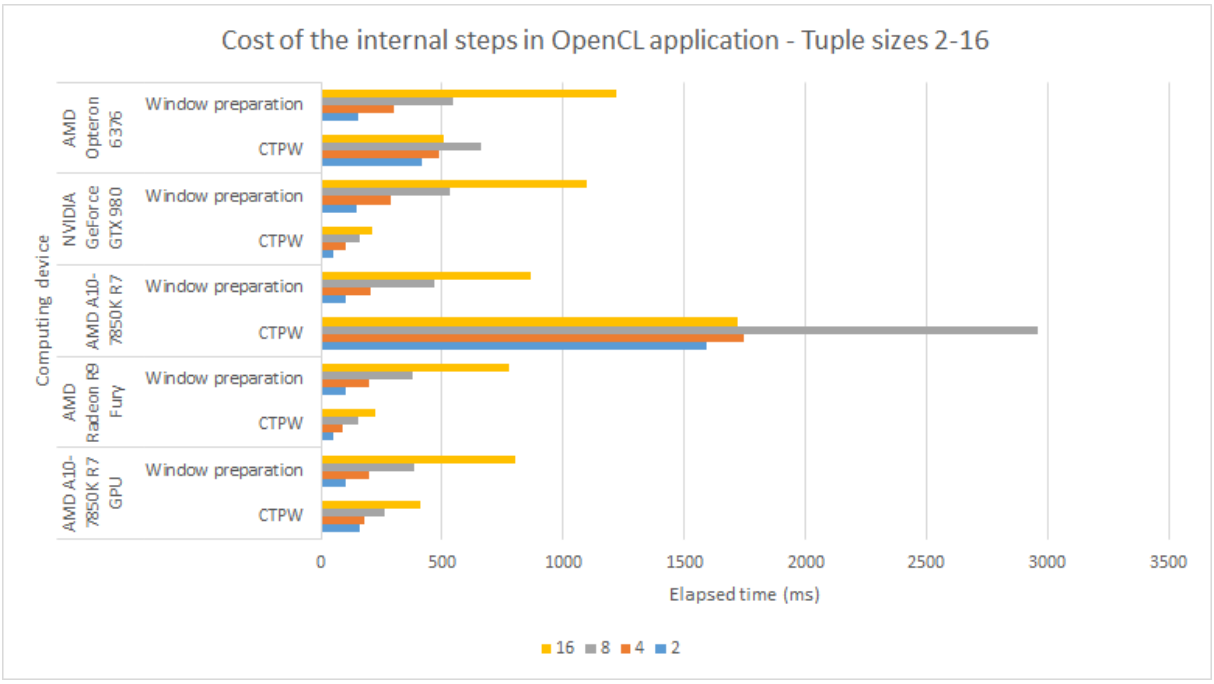

Figure 37: Cost of the internal steps in OpenCL application, tuple size 2-16

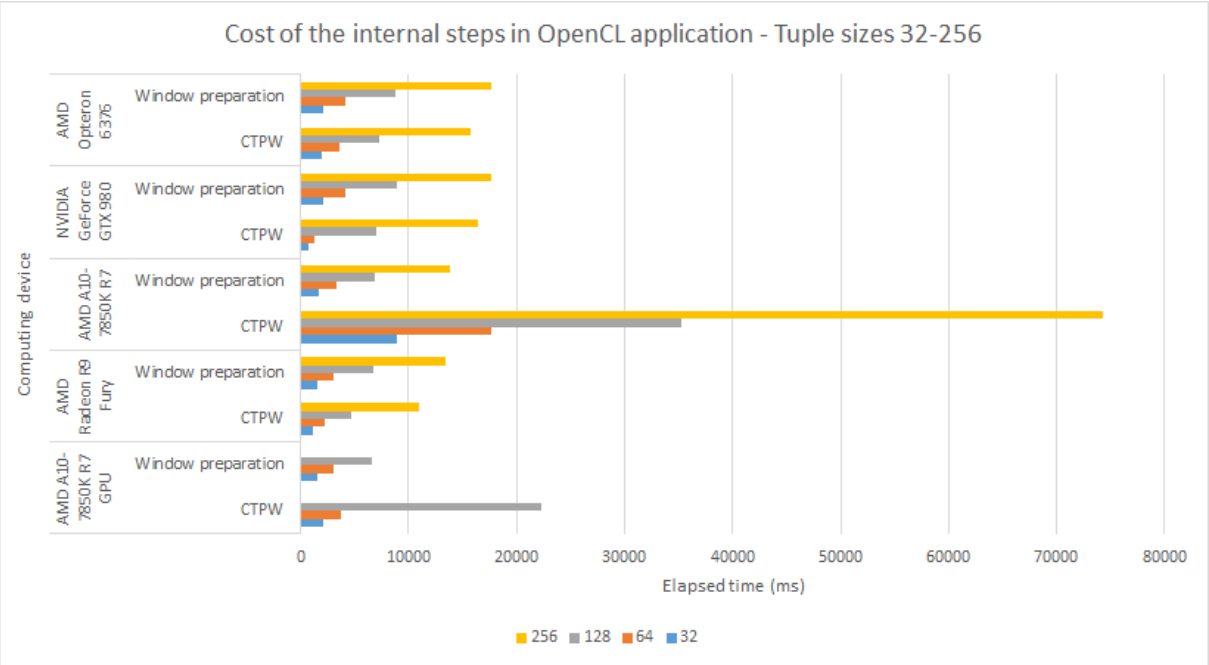

Figure 38: Cost of the internal steps in OpenCL application, tuple size 32-256

B.3. Cost of the Internal Steps in the Cluster Count Experiments

The figures below give the average elapsed times for window preparation and computational phases in the number of cluster experiments conducted in our OpenCL application.

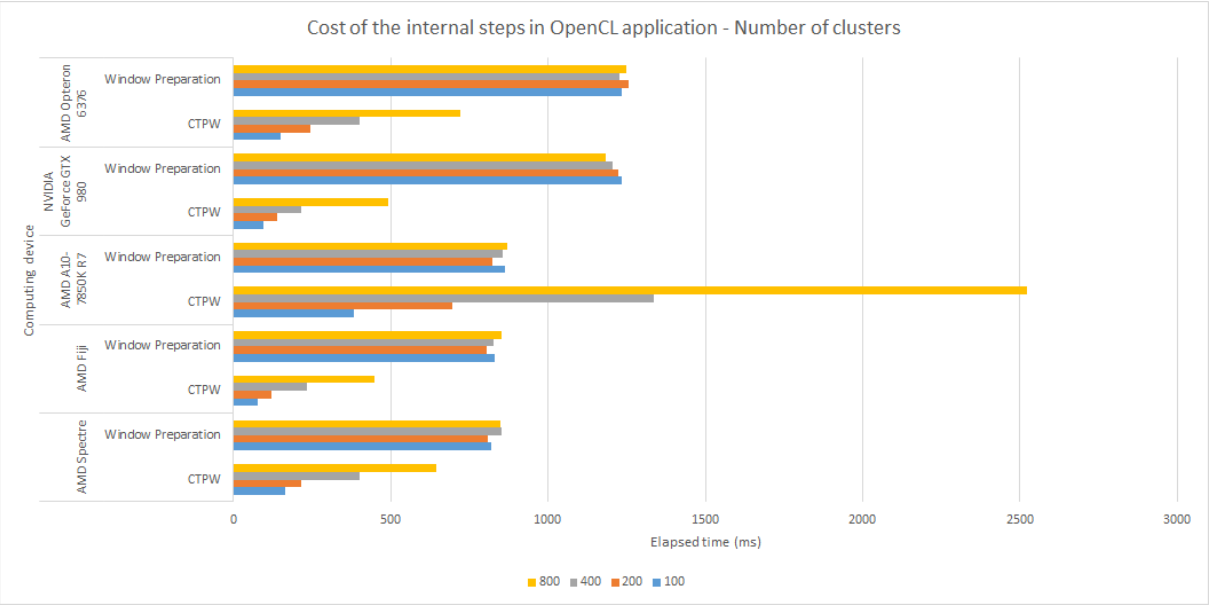

Figure 39: Cost of the internal steps in OpenCL application, number of cluster experiments

C. Additional Results for Apache Flink Experiments

The Figures below gives the cost of the internal steps of k-means window in which we implement streaming k-means algorithm in the Flink program.

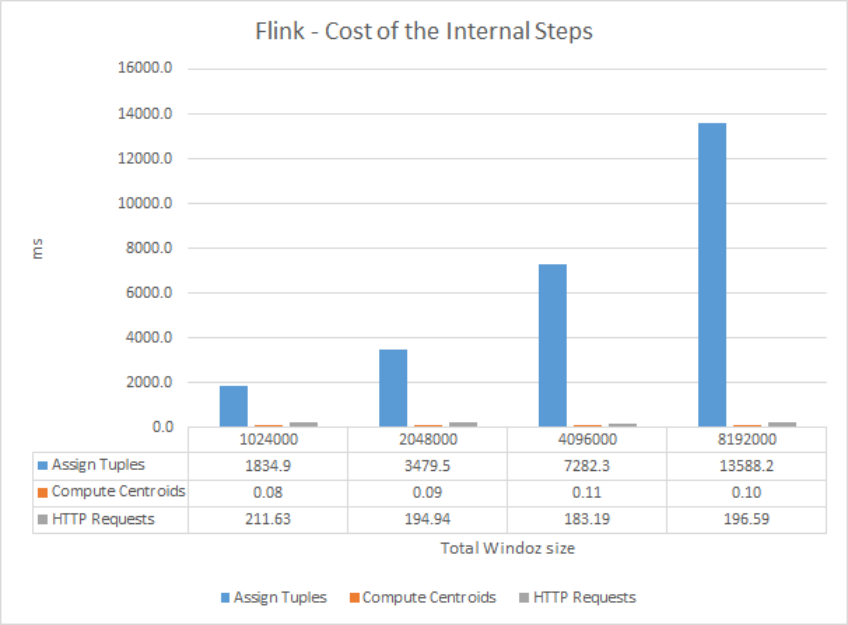

Figure 40: Cost of the internal steps in Flink application, the window size experiments

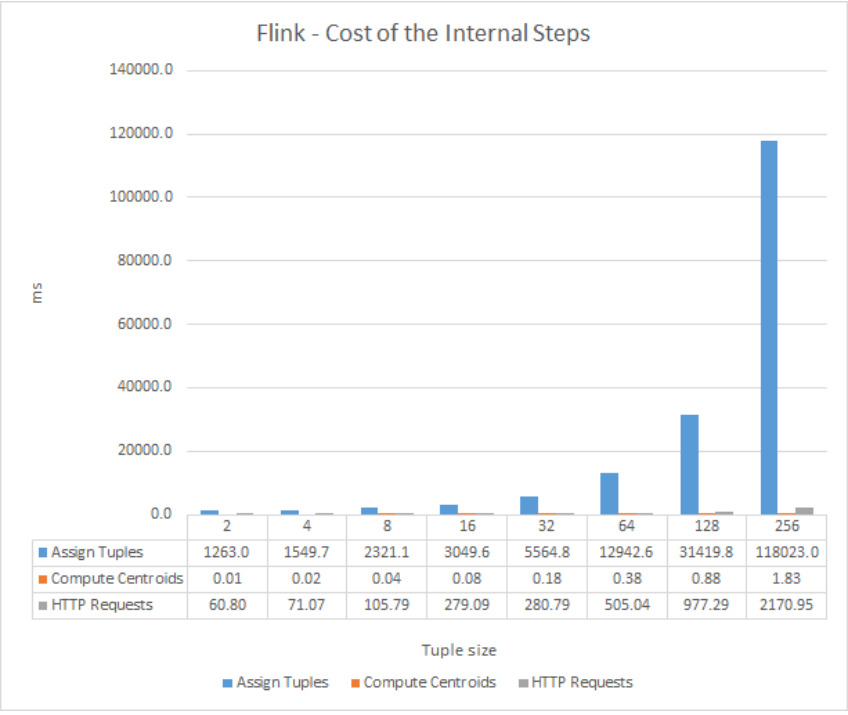

Figure 41: Cost of the internal steps in Flink application, the tuple size experiments

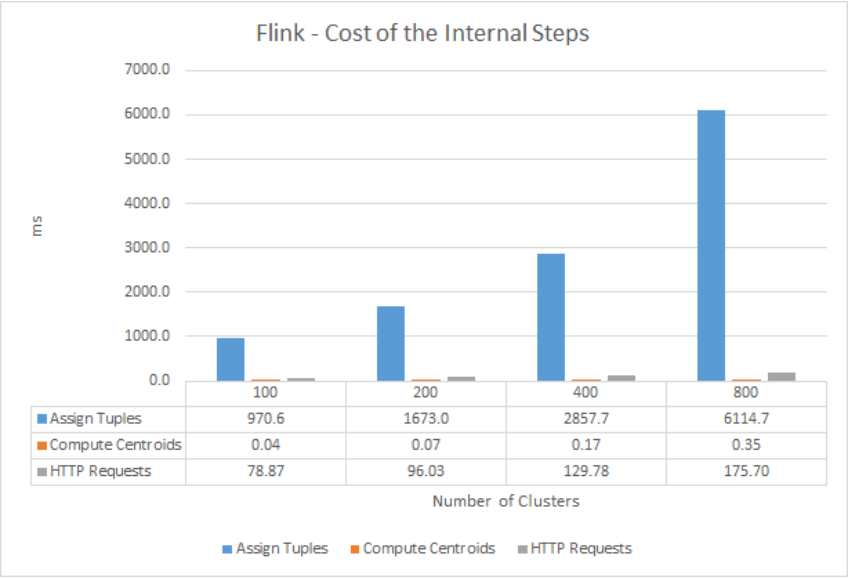

Figure 42: Cost of the internal steps in Flink application, the number of clusters experiments

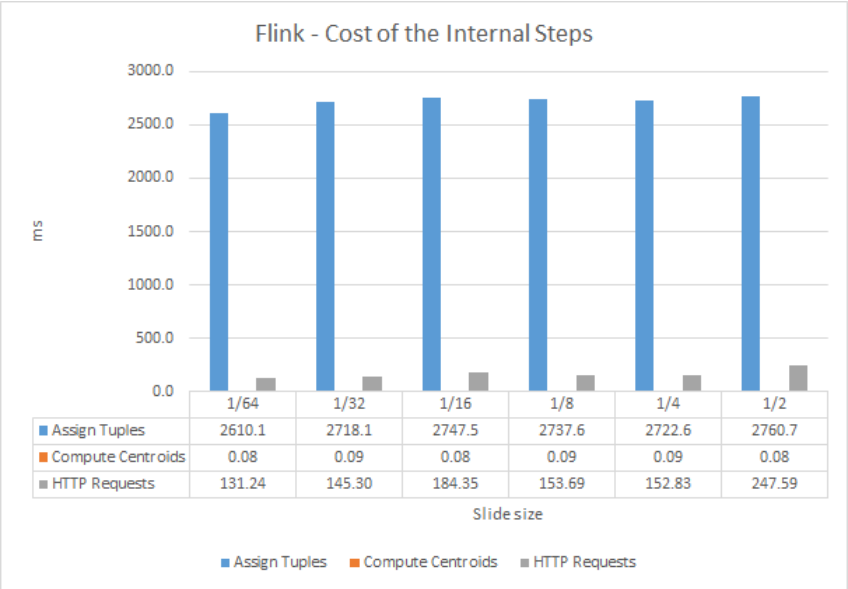

Figure 43: Cost of the internal steps in Flink application, window slide size experiments

D. Statistics

D.1. Euclidean Distance

Euclidean distance metric is given in the equation below. It is mostly used in K-means clustering algorithm and tend to detect hpyerspherical clusters [60].

$$D_{ij} = \left(\sum_{l=1}^{d} |x_{il} - x_{jl}|^{1/2} \right)^2 \tag{6}$$

D.2. Gaussian Distribution

Probability density function of Gaussian(Normal) Distribution is given below. Where;

- ϕ denotes the probability density function
- μ the mean
- σ^2 the variance of the distribution of x.

$$\phi\left(x|\mu,\sigma^2\right) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7}$$