K-MLIO: Enabling K-Means for Large Data-sets and Memory Constrained Embedded Systems

Camélia Slimani
Univ Brest
Lab-STICC, CNRS, UMR 6285
F-29200 Brest, France
Camelia.Slimani@univ-brest.fr

Stéphane Rubini
Univ Brest
Lab-STICC, CNRS, UMR 6285
F-29200 Brest, France
rubini@univ-brest.fr

Jalil Boukhobza
Univ Brest
Lab-STICC, CNRS, UMR 6285
F-29200 Brest, France
boukhobza@univ-brest.fr

Abstract—**Machine Learning (ML) algorithms are increasingly used in embedded systems to perform different tasks such as clustering and pattern recognition. These algorithms are both compute and memory intensive whilst embedded devices offer lower hardware capabilities as compared to traditional ML platforms. $K-$means clustering is one of the widely used ML algorithms. In the case of large data-sets, our analysis showed that on average, more than 70% of the execution time is spent on I/Os. In this paper, we present a version of $K-$means that drastically reduces the number of I/Os by spanning the data-set only once as compared to the traditional version that reads it several times according to the number of iterations performed. Our evaluation showed that the proposed strategy reduces the overall execution time on large data-sets by 60% on average while lowering the number I/Os operations by 90% with a comparable precision to the traditional $K-$means implementation.**

Index Terms—**K-means, I/O optimization, embedded systems, machine learning**

I. Introduction

Embedded systems are used in several fields [13], such as transportation, environment and health. Applications performed by these devices are more and more taking advantage of the impetus given by machine learning (ML) algorithms. For instance, in sensor networks, ML is used for anomaly detection [13] to address several attacks such as wormhole, sinkhole, and misdirection.

The data volume to process in embedded devices is experiencing an exponential growth [18] [20] while the main memory size in those devices is stagnating because of technology issues [14]. Then, the collected data are inevitably stored and swapped-in and out from the storage system for processing. This dramatically increases the processing time for large data-sets, thus the energy.

One of the most used clustering algorithms for embedded systems is the K-means [12]. It partitions a set of elements into K clusters each of which is identified by a centroid. Each element belongs to the cluster with the nearest centroid. To do so, K-means starts by selecting K points as initial centroids. The remaining points are affected to their nearest centroid in order to form the K clusters. Once done, the mean of each cluster is calculated to identify the new centroids. The process is repeated until the centroids do not vary between iterations.

This project is financially supported by the Bretagne region, France.

One major drawback of K-means is that the data-set is entirely scanned during each iteration, and there are as many iterations as necessary to converge to the final solution. When the data-set is fully stored into the main memory this is not a real issue. However, when it is several times larger than the available memory work-space, the number of generated I/O operations increases dramatically which makes the algorithm I/O-bound. Our experiments have shown that the time spent in I/O operations grows from 2% of the execution time for a data-set which size is half the memory work-space up to 70% when the data-set size exceeds the memory work-space size.

Several state-of-the-art studies have been proposed to enhance the K-means algorithm performance [4], [8], [11]. These methods aim mainly at reducing the number of computations performed. Some others [6] [10] propose to adapt K-means to large data-sets. In these methods, K-means is applied on small individual partitions, then the obtained solutions are re-clustered to get the final K centroids. These methods imply some precision loss. Mini-Batch K-means [19] proposed to reduce the number of computations by using only on a data subset during each iteration. One consequence is I/O cost reduction, but at the expense of a precision loss.

In this paper, we aim to reduce the I/O-cost of the K-means algorithm without compromising on the precision. The main idea of K-MLIO (for K-Means with Low I/O Overhead) is to decorrelate the number of data-set full scans from the number of iterations performed in order to scan the data-set only once, thus reducing drastically the I/O cost. To do so, we first divide the data-set into chunks that can fit into the main memory work-space, and perform independent K-means clustering on each of them. Then, we form a single representative chunk for the whole data-set. This is done by performing similarity analysis on the partial K-means results from the first step and grouping similar cluster through the chunks. Finally, K-means is run on the representative chunk. This way, regardless of the data-set size, the memory work-space size, and the data distribution, the data-set is read only once.

We compared K-MLIO to both the traditional K-means and Mini-Batch in terms of execution time and precision with different data-sets. The results show that K-MLIO reduces the execution time by 35-80% as compared the traditional K-means for large data-sets with a comparable precision. As

Notation	Description
K	Number of clusters
D	data-set
N	Number of elements in the data-set
M	Number of elements contained in memory
x	An element of the data-set
d	Number of dimension of an element
s	Size of a data-set element
S	Chunk size
B	Size of an I/O block
I	Number of iterations
i/o$_{cost}$	I/O cost per iteration
I/O$_{cost}$	Overall I/O cost
$E_{I/O}$	I/O cost enhancement
S_b	Number of elements per randomization block

TABLE I: Notation table

compared to Mini-Batch, K-MLIO proved to be more precise for all the experiments given a similar execution time. Another interesting result is that K-MLIO ran with a stable I/O cost (less than 4% of the total execution time) regardless of the data-set size and memory work-space while for the traditional K-means, the larger the data-set size for a given memory work-space, the higher the I/O cost.

The remainder of the paper is as follows: the second section gives more details about K-means and its I/O pattern. Section III describes the proposed method and presents its I/O theoretical analysis. Section IV exposes experiments to show the efficiency of the proposed method. Finally, section V concludes the paper and draws up some perspectives.

II. Motivation : I/O analysis of K-means

To motivate our study, we present a theoretical analysis of the I/O cost induced by the K-means execution, according to the data-set size ($N \cdot s$) and the allocated memory work-space ($M \cdot s$). Then, we point out, by an experimental analysis, the impact of N/M, the ratio between data-set and memory work-space sizes on the I/O time proportion. Table I summarizes the notations used in the rest of this paper.

1) Theoretical Analysis: Here, we suppose a simplified system architecture that is composed of a RAM memory work-space and a secondary storage space. The data-set to cluster is composed of N elements and the memory work-space can contain up to M elements. Memory work space size is $S = M \cdot s$ where s is the size of an element x_i.

For each iteration the full data-set is scanned. If the data-set is large ($N >> M$), then it is loaded into memory chunk by chunk (a chunk being equal to the memory work-space size) for each iteration. If the I/O block size is B, the I/O cost for each iteration caused by swap-in and out would be (note that no cache effect is considered here):

$$i/o_{cost} = \lceil N/M \rceil \cdot \lceil M/B \rceil \cdot B \quad (1)$$

In Eq. (1), $\lceil N/M \rceil$ represents the number of chunks in the data-set . $\lceil M/B \rceil$ is the number of blocks per chunk.

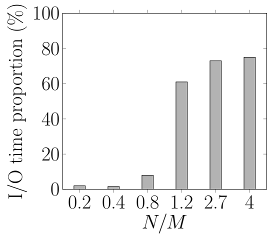

Fig. 1: Proportion of execution time spent on I/O operations for different data-set sizes N over memory work-space M

If the algorithm performs I iterations to converge, the overall I/O cost would be:

$$I/O_{cost} = \lceil N/M \rceil \cdot \lceil M/B \rceil \cdot B \cdot I \quad (2)$$

As mentioned in [21], it is established that the theoretical upper bound of I has an exponential complexity according to N and K. Then, I/O_{cost} has an exponential upper bound complexity. In practice, since the number of iterations is limited to a known bound, this complexity is bounded.

From Eq. (2), it appears that in the traditional K-means algorithm, the I/O volume swapped-in and out between the memory and the storage device depends on both the volume of the data-set and the convergence speed given by I. Thus, as $\lceil N/M \rceil$ grows, the I/O cost increases, which would increase the proportion of K-means execution time spent on I/Os. The experimental part in the next section confirms this behavior.

2) Experimental Analysis: In this section, we show the proportion of execution time spent in I/O operations performed by K-means for growing values of N/M. To do so, we used synthetic data-sets of different sizes (from 50MB to 1GB) and a fixed memory work-space of 256MB. More details about data-set generation are given in the evaluation part. The tests were performed on a BeagleBone Black embedded board.

Fig. 1 shows the execution time proportion spent on I/O operations. We observe that for data-sets that are larger than the available memory space, 72% of the overall execution time is spent on I/O operations. We conclude that since the data-set is fully scanned for each iteration, when it becomes larger than the memory work-space, the system needs to swap-in and out the whole data-set several times making the algorithm I/O-bound. The larger N/M, the worse is the performance.

III. K-MLIO, K-Means with Low I/O Overhead

In this section, we present a method for enabling **K-M**eans with a **L**ow **I/O** overhead for large data-sets and memory constrained embedded system (K-MLIO). We start by giving an overview of this algorithm, then, each step of the proposed method is detailed. We give a theoretical I/O analysis of K-MLIO and finally, discuss some optimizations performed on K-MLIO to cope with some particular cases.

A. Overview of the method

The main idea behind K-MLIO is, first, to partition the data-set into chunks that can fit into the main memory, and then

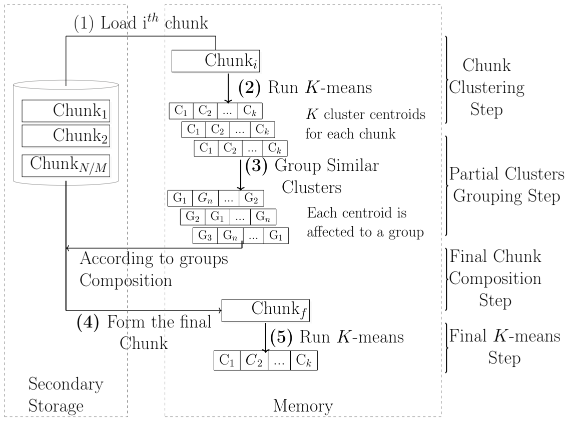

Fig. 2: K-MLIO overview

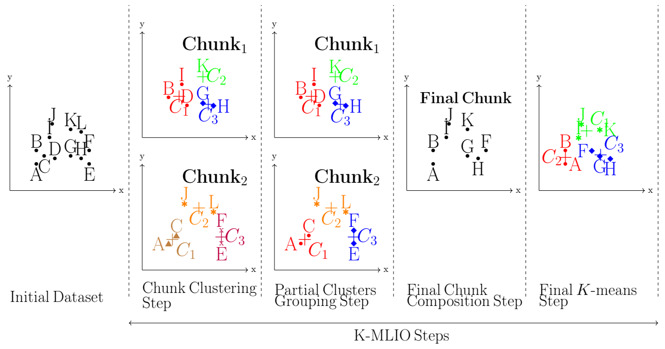

Fig. 3: K-MLIO execution example

apply K-means on each chunk separately until convergence to get K centroids for each chunk. Secondly, we combine these solutions to form a representative chunk for the whole data-set using similarity analysis. Similarity between clusters reflects that their associated chunks contain elements that are likely to belong to the same final cluster. This information is useful to form a representative chunk by selecting proportionally elements from the clusters of different chunks from different groups. Finally, K-means is run on this representative chunk in order to find the final K centroids for the whole data-set. With this method, each chunk is only loaded once into the memory, regardless of the number of iterations.

Fig. 2 shows the different steps of the K-MLIO algorithm:

Chunk Clustering Step: Each chunk is loaded into memory to run the K-means on. This results in K centroids for each chunk, see Fig. 2 (1 and 2);

Partial Clusters Grouping Step: The objective of this step is to group similar clusters. This allows to have an overview about the distribution of the elements that might belong to the same cluster among the chunks. As shown in Fig. 2 (3), each partial centroid is affected to a group;

Final Chunk Composition Step: The final chunk is built from representative elements according to the overall group composition. The size of this sample chunk is determined to be equal to the memory work space, see Fig. 2 (4);

Final K-means computation: We apply the K-means algorithm on the final chunk to get the final clustering solution for the data-set, see Fig. 2 (5).

B. *K-MLIO Algorithm Description*

In this section, each step of the proposed method is detailed. Fig. 3 shows an example with a data-set composed of 12 2D-elements (A to L), the number of clusters to define is $K = 3$. We rely on this example to explain the different steps.

1) Chunk Clustering Step: For a data-set D of N elements, if the available memory work-space can contain M elements, the number of chunks to cluster is $\lceil N/M \rceil$. For each chunk, we apply traditional K-means until convergence. The outputs are both the K centroids and the element assignment, which associates to each element x_i, its closest centroid.
The objective of this step is to avoid swap-in and swap-out of the chunks at each iteration, only the results need to be maintained in the main memory. In Fig. 3, the initial data-set is split into two chunks. Each chunk is clustered separately, the results are the centroid sets $\{C_1, C_2, C_3\}_1$ and $\{C_1, C_2, C_3\}_2$, and the element assignment.

2) Partial Clusters Grouping Step: In this step, similarity analysis is performed to detect similar clusters across the chunks. This is done to get an overview of the distribution of the potential final clusters on the data-set. Two clusters are defined to be similar if the distance between their respective centroids is smaller than the variance between the elements and their respective centroid within each of the two clusters.

The number of formed clusters in the first step is $\lceil N/M \rceil \cdot K$. Theoretically, when the data are uniformly distributed across the data-set (i.e. each chunk has a balanced number of elements from the final clusters), the number of formed groups would be equal to K. However, in the worst case, the number of groups could be equal to the number of partial clusters $\lceil N/M \rceil \cdot K$, where each chunk represents a distinct cluster.

In Fig. 3, the clusters associated to the centroid C_1 of chunk 1 and C_1 of chunk 2 are similar since the distance between the two centroids is smaller than the variances inside the two respective clusters. Then, the two clusters are grouped (simlar colors in the Figure). At the end of this step, we have 4 resulting groups: $G_1 = \{A, C, B, I, D\}$, $G_2 = \{J, L\}$, $G_3 = \{K\}$ and $G_4 = \{F, E, G, H\}$.

This step gives an idea about data distribution on the data-set. For instance, B and C that are in different chunks are likely to belong to the same final cluster.

3) Final Chunk Composition: In this step, given the previous data partitioning into groups, we form one representative chunk for the whole data-set. To do so, we randomly select from each group a subset of elements proportional to the number of elements it contains with regards to the total number of elements N. If the number of elements in a group G_i is N_i, then the number of elements to select from G_i is:

$$Sel_i = \lceil N_i/N \cdot M \rceil \quad (3)$$

In Fig. 3, three elements are selected from the first group G_1: A, B and I. One element is selected from the group G_2: J. K is the only element selected from group G_3. Three elements are selected from group G_4: G, H and F. Note that in this example, 8 elements are used to form the representative chunk out of 12, which does not bring really any performance enhancement. This configuration is unrealistic and only aims to help explaining the algorithm. In reality, the final chunk has always the same size (the size of memory work-space), regardless of the size of the data-set.

4) Final K-means computation: In this step, K-means is run on the previously formed representative sample chunk (resident in the main memory work-space). The resulting centroids represent the final solution to the data-set clustering.

C. I/O Analysis for K-MLIO

As we did for the traditional K-means, we analyze hereafter the theoretical I/O cost induced by the proposed method. From the four steps of K-MLIO, only step 1 and 3 are I/O costly. Indeed, the similarity analysis based on the variance calculations is precomputed for each chunk during step 1. Thus in step 2, the partial grouping is performed without accessing data. In step 4, the final chunk is already in memory and calculations do not require I/O operations.

Step 1 and 3 imply the following I/O cost:

Chunk clustering (step 1): Each chunk is loaded from the storage device once, so the I/O cost is:

$$I/O_{cost}^{(1)} = \lceil N/M \rceil \cdot \lceil M/B \rceil \cdot B \tag{4}$$

Final Chunk Composition (step 3): Here, M elements are selected from the initial data-set. An index is created to store the location of the chunks on the storage device. In addition, the elements to select are grouped by chunk in order to load each chunk only once. So, in the best case, where all the elements to select are in the same chunk, the I/O cost is:

$$I/O_{cost}^{(2)} = \lceil M/B \rceil \cdot B \tag{5}$$

In the worst case, where each chunk may contain some elements to select, the I/O cost is:

$$I/O_{cost}^{(2)} = \lceil N/M \rceil \cdot \lceil M/B \rceil \cdot B \tag{6}$$

The overall I/O cost of the proposed method is:

$$I/O_{cost} = I/O_{cost}^{(1)} + I/O_{cost}^{(2)} \tag{7}$$

Then:

$$\left(\left\lceil \frac{N}{M} \right\rceil + 1\right) \cdot \left\lceil \frac{M}{B} \right\rceil \cdot B \leq I/O_{cost} \leq 2 \cdot \left\lceil \frac{N}{M} \right\rceil \cdot \left\lceil \frac{M}{B} \right\rceil \cdot B \tag{8}$$

Eq. (8) shows that, contrary to the traditional K-means, the I/O cost of K-MLIO does not depend on the number of iterations.

Then, K-MLIO I/O reduction over K-means is:

$$E_{I/O} = \frac{I/O_{cost}^{(T)} - I/O_{cost}}{I/O_{cost}^{(T)}}$$

Where $I/O_{cost}^{(T)}$ is the I/O cost of traditional K-means. Using Eq. (2) and (8), we get the following expression for the theoretical optimization achieved:

$$1 - \frac{2}{I} \leq E_{I/O} \leq 1 - \frac{1}{I} \cdot \left(1 + \frac{1}{\lceil N/M \rceil}\right) \tag{9}$$

In Eq. (9), we observe that the optimization achieved depends on the data-set size with regard to the available memory space $\lceil N/M \rceil$ and the number of iterations I necessary for K-means to converge. The higher these two parameters, the better the efficiency of K-MLIO as compared to traditional K-means.

D. K-MLIO special cases

We have identified two cases where K-MLIO could not be helpful and designed techniques to address such cases.

1) K-MLIO on small data-sets: K-MLIO was designed to be effective when $\lceil N/M \rceil$ is large, meaning that the data-set is larger than the main memory work-space. In case the data-set can fit into the memory, K-MLIO performs poorly as it experiences a higher CPU load than traditional K-Means. In order to solve this issue, we adapted K-MLIO to run its I/O optimizations only in case the data-set size is larger than M. Otherwise, the traditional K-means is run.

2) The case of Non-uniform data-sets: In the case of a non-uniform distribution, each chunk may contain mainly elements from the same final cluster. In such a configuration, each chunk would require a high number of iterations to converge since the elements it contains are close to each other. Thus, the partial clustering step would be costly in terms of computations. However, this limitation does not affect the solution precision since the partial cluster grouping step allows eventually to sample a representative chunk.

K-MLIOR, K-MLIO with Randomization: To address this issue, we propose to randomize the data before executing K-MLIO in such a way to avoid having a high proportion of elements of the same cluster within the same chunk.

We define a block size S_b used for the randomization process. Each chunk is then composed of $\lceil M/S_b \rceil$ blocks randomly chosen from the data-set. As random reads are efficient on flash memory [2], randomization is cheap.

To randomize the data-set, we modified the first step of K-MLIO. We used an index to maintain the association between randomly read blocks for each chunk. During the 1^{st} step of K-MLIO, the blocks to load are read with the help the index.

IV. Evaluation

A. Experimental Methodology

K-MLIO and K-MLIOR were compared to both traditional K-means algorithm and Mini-Batch K-means.

1) Experiments: To evaluate the relevance of K-MLIO, we performed the following experiments:

Experiment 1 was performed to evaluate the efficiency of K-MLIO as compared to K-means for different N/M values. The evaluation metrics are detailed in the next section. We used different data-set sizes: 100MB, 400MB, 700MB and 1GB which respectively represent these N/M proportions :

0.4, 1.5, 2.7 and 4 for a 256MB memory size. We have also varied the separation index between clusters [16]. The higher this value, the lower the number of iterations necessary to converge. We performed experiments on three configurations: highly, moderately and slightly separated clusters.

Experiment 2: The objective of this experiment is to compare K-MLIO to Mini-Batch. As Mini-batch proved to have precision issues and in order to achieve relevant comparison, we measured the precision obtained by both algorithms for the same execution time. To do so, we measured the execution time of K-MLIO and fixed the number of iterations of Mini-Batch in a way to have the same execution time. We performed this experiment with different N/M and with a $sepval = 0.01$.

Experiment 3: It is performed to show the advantage of using K-MLIOR when it comes to non-uniform distributions. We tested K-means, K-MLIO and K-MLIOR with different N/M. The tests were performed for an extreme case where the data-set is fully ordered, ie. all the elements of a cluster are in the same chunk sequence. The data-sets were obtained by ordering previously generated ones. The randomization block size was set to $S_b = 512$ elements.

Experiment 4: It allows us to observe the I/O time proportion of K-MLIO, as compared to traditional K-means.

2) Measured Metrics: We measured the execution time, the clustering Average and the I/O time proportion throughout our experiments. For the clustering Average, we calculated the average distance error between the obtained solution and the correct one, by forming K pairs of centroids (C_i, c_i), where C_i is the obtained centroid and c_i is the correct one. As in [15], the error was measured as follows:

$$e_i = \frac{1}{d} \cdot \sum_{l=1}^{d} |c_i^l - C_i^l| / |c_i^l| \tag{10}$$

The average error for the K centroids is then :

$$E = \frac{1}{K} \cdot \sum_{i=1}^{K} e_i \tag{11}$$

We considered that the average error is acceptable if ($E \leq 1$).

3) Experimental Data-sets: We performed the previously described experiments on synthetic data-sets generated using ClusterGeneration package of the R platform [17]. This package allows to generate a data-set of N elements that belong to K clusters in a given dimension. We set the number of dimensions to 10. Cluster separation index is between -1 and 1: The closer to 1, the more the clusters are separated. We set the index to 0.6 for separated clusters, 0.01 for moderately separated clusters and -0.6 for slightly separated clusters.

4) Hardware platform: We used a BeagleBone Black board [5]. The main memory size is 512MB, the processor is AM335x 1GHz ARM Cortex-A8. We used a 2 GB partition on an SD card to use as a swap area.

B. Results

Experiment 1, K-MLIO vs K-means: For the first experiment, we show the execution time reductions when the error is

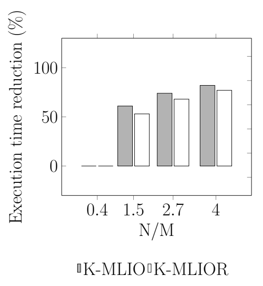

Fig. 4: $sepval = 0.01$

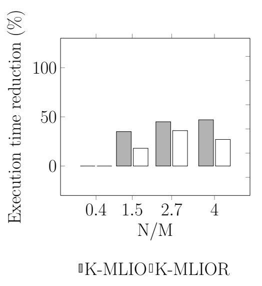

Fig. 5: $sepval = 0.6$

K-MLIO and K-MLIOR execution time reduction as compared to K-means.

Sepval	N/M	Error	
		K-means	K-MLIO
-0.6	0.4	6.54	6.54
	1.5	5.7	5.9
	2.7	3.86	3.93
	4	6.2	5.28
0.01	0.4	0.22	0.22
	1.5	0.24	0.15
	2.7	0.11	0.12
	4	0.12	0.12
0.6	0.4	0.003	0.003
	1.5	0.0014	0.007
	2.7	0.0007	0.0036
	4	0.003	0.01

TABLE II: Average Errors for Experiment 1

acceptable. Table II shows the average error for K-means and K-MLIO. We observe that for $sepval = -0.6$, both methods did not converge as the clusters are too close to each others. Figures 4 and 5 show the execution time enhancement for $sepval = 0.01$ and $sepval = 0.6$. We observe that K-MLIO and K-MLIOR reduce the execution time respectively by 60-82% and 50-80% for $sepval = 0.01$, and 35-50% and 20-40% for $sepval = 0.6$ as compared to K-means. Execution time reductions are lower for $sepval = 0.6$ since both algorithms need to perform a smaller number of iterations to converge, thus less I/Os for the K-Means; Eq. (9) shows that when the number of iterations is small, I/O enhancement is low.

We observe from Table II that K-MLIO and K-means have comparable errors. Note that all errors were calculated compared to the correct (known) centroids. For N/M value of 0.4, as K-MLIO runs the K-means without optimizations since the memory work-space is large enough to include all the data-set, the error was the same.

N/M	Average Error	
	K-MLIO	Mini-Batch
0.4	0.22	0.6
1.5	0.15	0.91
2.7	0.12	0.58
4	0.12	0.59

TABLE III: Average errors for Experiment 2

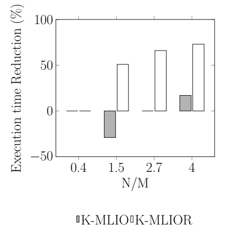

Fig. 6: Execution time reductions as compared to K-means

N/M	Error		
	K-means	K-MLIO	K-MLIOR
0.4	0.22	0.22	0.22
1.5	0.2	0.15	0.15
2.5	0.11	0.12	0.12
4	0.12	0.18	0.12

TABLE IV: Average errors for Experiment 3

Experiment 2, K-MLIO vs Mini-Batch precision: Table III shows the results for the second experiment. We observe that K-MLIO errors were always lower than that of Mini-Batch. This is because Mini-Batch considers only partial subsets of data to compute the K-means, and it is not particularly optimized for I/Os. So when K-MLIO spans all the data-set once to compute all the K-means for the different chunks, Mini-batch partially scans some chunks several times.

Experiment 3, advantage of K-MLIOR with non-uniform data-set distribution: Fig. 6 shows the execution time reductions obtained with the tested algorithms. We observe that K-MLIOR achieves higher reductions. K-MLIO proved to have higher execution time where N/M had a low value (still higher than 1). As expected, K-MLIO performs a high number of iterations to converge since the elements in the chunks are close to one another. However, when N/M grows, this high number of iterations is compensated with the high I/O reduction achieved. Note that the data were fully ordered which represents the worst case distribution for K-MLIO from a CPU computation point of view. Table IV shows the average errors obtained. We observe that in terms of precision K-means, K-MLIO and K-MLIOR are comparable.

Experiment 4, I/O operations reduction of K-MLIO and K-MLIOR: Table V shows the reduction of the proposed algorithms in terms of I/Os. We observe that I/O time proportion for K-MLIO does not exceed 3.5% for all the tested data-sets, whereas it reaches 75% for traditional K-means for high N/M values ($N/M = 4$). Overall, we reduced the number of I/O operations by more than 90% in all tested cases. Note that K-MLIO and K-MLIOR perform the same number of I/O operations, the difference between both is related to the randomization process which does not infer additional I/Os.

V. Related work

Studies in [6] and [10] propose to perform clustering on large data-sets by applying K-means on individual partitions of the data-set as a first step, then applying a final K-means round on the obtained partial clusters to get the global solution. This method induces a precision loss due to the fact that the number of points to cluster is reduced. To address this issue, the authors of both studies proposed solutions that imply whether a slower clustering or more reload operations. One side effect of these algorithms is the reduction in I/Os.

In [9], the authors propose to execute K-means on one randomly sampled small subset, then the centroids are adjusted using the data-set points that are likely to be affected to different centroids. This method depends on the data distribution and implies to span the data-set to detect those points until adjusting the centroids. Other studies [7] [19] proposed to make K-means algorithm adaptive to massive data-sets by executing iterations of Lloyd's algorithm on smaller data-sets. Mini-Batch K-means [19] proposes to randomly sample the data-set to compose a batch at each iteration. Then, the elements of the batch are affected to the nearest centroids. The new centroids are calculated using gradient-descent method [1]. The number of iterations to perform is fixed. This method is efficient for reducing K-means execution time for large data-sets. One consequence of this method is I/O cost reduction. However, this is done at the expense of precision loss.

N/M	I/O time (%)		K-MLIO I/O time reduction (%)
	K-means	K-MLIO	
0.4	1.6	1.6	0
1.5	67	3.5	94
2.7	73	3.5	95
4	75	3	96

TABLE V: I/O Time reduction for Experiment 4

VI. Conclusion and Future Work

This paper proposes a method to enable K-means for large data-sets in embedded systems with a restrained memory space. When the data-set is larger than the main memory work-space, the traditional K-means generates a significant I/O overhead as it scans the data-set several times. The proposed solution consists of loading chunks that fit into the memory work-space and applying K-means on each of them. The results are combined so as to produce a representative sample to run final data-set clustering solution on. Our experiments show that this method allows to reduce K-means algorithm execution time up to 80% while keeping a comparable precision. Finally, additionally to embedded system applications, K-MLIO could also be used for other applications. For instance, in the case of a Cloud service in which one would like to reduce the costs and thus reduce the resource usage. In this context, K-MLIO could be used to reduce the memory footprint of a container or a virtual machine.

For future work, we would like to investigate emerging Non-Volatile-Memory [3] integration in embedded systems and how we can take benefit from such a technology. We also would like to study other widely used ML algorithms such as random forests.

References

[1] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 585–592. MIT Press, 1995.

[2] J. Boukhobza and P. Olivier. *Flash Memory Integration: Performance and Energy Issues, 1st Edition*. ISTE Press - Elsevier, 2017.

[3] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):14:1–14:32, Nov. 2017.

[4] M.-C. Chiang, C.-W. Tsai, and C.-S. Yang. A time-efficient pattern reduction algorithm for K-means clustering. *Information Sciences*, 181(4):716 – 731, 2011.

[5] G. Coley. *BeagleBone Black System Reference Manual*. 2013.

[6] H. Cui, G. Ruan, J. Xue, R. Xie, L. Wang, and X. Feng. A collaborative divide-and-conquer k-means clustering algorithm for processing large data. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 20:1–20:10, New York, NY, USA, 2014. ACM.

[7] I. Davidson and A. Satyanarayana. Speeding up k-means clustering by bootstrap averaging. In *IEEE data mining workshop on clustering large data-sets*, 2003.

[8] C. Elkan. Using the triangle inequality to accelerate K-means. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pages 147–153. AAAI Press, 2003.

[9] A. Goswami, Ruoming Jin, and G. Agrawal. Fast and exact out-of-core k-means clustering. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 83–90, Nov 2004.

[10] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, May 2003.

[11] G. Hamerly. Making K-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 130–140. SIAM, 2010.

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

[13] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth. Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*, 4(3):161 – 175, 2018.

[14] O. Mutlu. Memory scaling: A systems architecture perspective, Aug. 2013.

[15] C. Ordonez and E. Omiecinski. Efficient disk-based K-means clustering for relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):909–921, Aug 2004.

[16] W. Qiu and H. Joe. *Random Cluster Generation (with Specified Degree of Separation)*. 2015.

[17] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[18] P. Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44(1):39–48, Jan 2011.

[19] D. Sculley. Web-scale K-means clustering. In *Proceedings of the* 19^{th} *International Conference on World Wide Web*, WWW '10, pages 1177–1178, New York, NY, USA, 2010. ACM.

[20] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future*, 16, 2014.

[21] A. Vattani. K-means requires exponentially many iterations even in the plane. *Discrete & Computational Geometry*, 45(4):596–616, Jun 2011.