Evaluating a *Processing-in-Memory* Architecture with the k-means Algorithm

Simon Bihel, Lesly-Ann Daniel, Florestan De Moor and Bastien Thomas
University of Rennes 1 / ENS Rennes
{first.last}@etudiant.univ-rennes1.fr

***Abstract*—Novel *big data* workloads like genomics or management of large databases are rising and have yet to be treated efficiently. The gap between processing and memory access has not been filled and processing power is currently increasing only because of parallelization. A shift from computing-centric to data-centric architectures is thus required and the concept of *processing in-memory* is a promising way of doing it. This paper evaluates this concept with the commonly used k-means algorithm. The platform used to implement it is the UPMEM architecture which is a DRAM with embedded computing capabilities. Through experiments, we show that the architecture's low computation power induces limitations for applications without a big memory bottleneck.**

I. Introduction

Over the last decades, the growth of the Internet and the creation of databases by various organizations have led to large volumes of information that need to be analyzed. Consequently, improving data-intensive applications became a main concern. The performances are not limited by the computational speed but rather by the memory access. Traditional architectures are vulnerable to memory bandwidth and latency issues. This slows down the computations by preventing processors from running at full speed. This is referred as the memory wall. Since increasing drastically the bandwidth does not seem to be easily achievable, trying to reduce the communications between the processors and the memory might be easier. The hardware solutions to this problem generally involve 'cache' memory which consists in bringing faster memories closer to the processor.

Another kind of solutions is *near-data processing* which consists in doing the opposite: bringing processing units close to the memory, or to the pathway used during data movement. The architecture which we are particularly interested in is the *processing in memory* (PIM) which integrates processing units into the main memory.

UPMEM [1] is a project aiming to produce a PIM architecture. This technology adds co-processing units into the DRAM that can execute operations on behalf of the CPU, while minimizing the data flow. A 16GBytes UPMEM-DIMM embeds 256 DRAM Processing Units (named DPU) and up to 16 of them can be added to a standard CPU. This model offers a massively parallel environment while reducing the latency, and increasing the bandwidth which makes it ideal for data-consuming operations. However, the application has to be adjusted to take advantage of this architecture.

This technology therefore seems to be a good solution to run data intensive tasks like clustering. Cluster analysis is used in a consequent number of applications, such as image segmentation, social network analysis, medical imaging, or gene sequence analysis. In particular, k-means clustering problem is well-known. The goal is to partition a set of points into k clusters while minimizing the sum of distances from each data point to the centroid of its cluster. Stuart Lloyd's algorithm [2] became popular because of its efficiency and is still widely used. The centroids are initialized randomly and each point gets the class of its nearest centroid. We then update the centroids and repeat the operation until a convergence criteria is met. This algorithm has a polynomial complexity, and it can return a local optimum, depending on the initialization of the first centroids. Finding an optimal solution is however NP-hard [3]. k-means++ algorithm [4] uses a randomized seeding technique to improve both runtime and clustering quality. This clustering problem introduces many distance computations, and those can be effectively parallelized.

This paper proposes an implementation of the k-means algorithm on the UPMEM PIM architecture.

The rest of this paper is organized as follows. Section II gives an overview of PIM architectures and of k-means enhanced algorithms. The architecture of UPMEM is developed in Section III. In Section IV, we describe our implementation of the k-means algorithm for the UPMEM architecture. We then evaluate it experimentally in Section V, and finally conclude in Section VI.

II. Related Work

II-A will first do a quick summary of what has been done on the topic of PIM. This will allow us to visualize the place of UPMEM in the ecosystem. Then II-B will go through previous work done on the k-means.

A. Processing In Memory

The concept of *near-data processing* (NDP) has been around since the 1960s [5]. Through the years, there has been more or less interest in it, following traditional computer architecture scaling and problems necessities. As we know with recent *big-data* challenges, we are facing walls of memory, bandwidth and power. A shift from compute-centric to data-centric computing systems is thus needed and the field of NDP, and particularly PIM, is blooming [6].

PIM is only a subset of NDP which can also include processing during transport (e.g. intelligent network, in front of bus/switch, ...), reconfigurable architectures [7], [8], processing in cache or on disk, ... Systems with such capabilities would allow the replacement of DRAM with non-volatile memory which offers more storage at a lesser cost and a lower power consumption. On a side note, NDP projects are often called PIM even though it is not really.

PIM is being focused on because it is the most radical architecture to tackle data processing problems. PIM allows massive bandwidth and low latency. Combined with specialized processing units it really speeds up the computing process. Reducing data motion also means less power consumption and leaving off-chip communication for other applications, resulting in a more efficient system.

2D-integrated PIM has been the first explored path. Embedding logic and memory onto the same die is functionally feasible, enabling very wide bus interfaces. This can be transparently used as regular DRAM and can be attached to external memory. One drawback is the unavoidable choice to make between high and efficient memory storage and fast logic.

Since the late 1990s, 3D integration of logic and memory layers has emerged. In addition to even greater bandwidth and lower latency due to on-bus bus frequencies being higher, the width of the bus is flexible. Freeing from pin constraints also means more pins available for more important work or push a little back the *power wall* as it is constrained by the important growth of supply pins compared to the total pins development. Memory management is also faster and more efficient due to the logic layer.

Shifting from the long established computing-centric architecture comes with big usability challenges. These problems evolve around the fact that there is no consortium on what a PIM architecture should be or what language/interactions it should have, and the dependency on vendors. Easy PIM programmability is key for wide adaptation, depending on the memory-view and usage. For 2D PIM, it depends only on the memory vendor which is not used to this kind of problems. This is not true for 3D PIM because of the separation of the layers but the flexible inter-layers bus makes it harder to program. While the possible transparency of 2D PIM allows a smooth architectural transition, a business case is required for vendors to commit in this area. Another commercial limitation is the absence of upgradeability because of the tight integration of memory and processing.

B. KMeans

We distinguish two main lines of research to improve the performances of the k-means clustering algorithm. The first focuses on the standard algorithm, and consists mostly in avoiding useless computations. These improvements can be included in both sequential and parallel implementations. The second line uses parallelism to obtain significant speed up.

1) KMeans Standard Algorithm Enhancements: [9] proposes an amelioration of the k-means algorithm using a data structure called a kd-tree (k dimensional tree). Each node of such a tree stores information on points contained in a hyper-rectangle. The root of the kd-tree is the rectangle containing all the points of the figure. Each rectangle is then recursively split in two 'balanced' rectangles. Generally, the rectangles are split along their longest side, so that the two resulting rectangles have the same number of points, but other splitting patterns are possible. In some cases, it can be easily known that a centroid is not the nearest class for every point of a node. This can then be used to avoid unnecessary computations. Despite the overhead introduced when building the data-structure, the algorithm shows significant speed-up, both on generated and real datasets.

In [10], the authors propose an enhanced k-means algorithm which improves the execution time. For each point, the distance to the current nearest cluster is kept in memory. At the next iteration, they first compute the distance to the new centroid of the same cluster. If is is less or equal to the distance kept in memory, then the point will not change cluster. It is thus not needed to compute the distance to the $k - 1$ other clusters. This algorithm based on the simple idea of benefitting from previous iterations avoids useless computations. Experimental results with several datasets show it is up to three times faster than the standard algorithm.

The authors of [11] propose an optimized k-means algorithm by reducing the data that is processed at each loop. They define a border area made of the points close enough to the edge of their cluster so that the next iteration could make them switch clusters. It is thus possible to limit the data space visited at each loop to this area, which greatly reduces the number of computations. Experiments with randomly generated datasets show a reduction of the running time from 50% to 70%.

[4] proposes a heuristic for the initial placement of the centroids. The first seed is initialized as a random point of the dataset. The next ones are chosen with probability that promotes points that are far away from seeds that are already chosen. Just after the initialization phase, there are already some probabilistic guaranties about the quality of the clustering, and those can only be improved with the k-means algorithm. In practice, this simple initialization improves significantly both the running time, with a speed-up from 2 to 10, and the quality of the solution, with at least a 10% accuracy improvement, both for random and real data-sets.

2) Parallel KMeans: [12] proposes a parallel k-means algorithm using MapReduce, a powerful parallel programmming technique. Hadoop was used as the MapReduce system for the experiments. Results show the algorithm has a great scalability: a speedup around 3 is obtained when using 4 nodes. The algorithm can thus process efficiently large datasets.

The authors of [13] propose MKmeans, a parallel clustering algorithm using MPI (Message Passing Interface). MPI is an API library which provides a standard programming model for message passing bewteen a set of processes. The algorithm is implemented using data parallelism: the points are split into n sets. Each of the n processes execute the sequential k-means algorithm on its dataset. A final merging function is used to

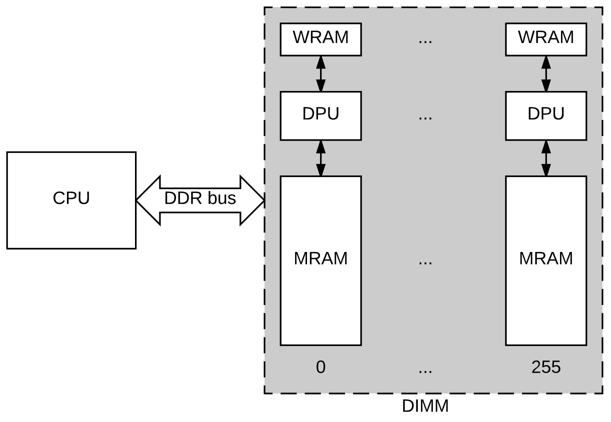


Fig. 1. **Illustration of UPMEM architecture**

generate k centroids from the $k * n$ centroids outputted by the n processes. An experimental evaluation is performed using datasets from the UCI Machine Learning Dataset Repository. Results show MKmeans improves the time performance of the clustering algorithm with large-scale datasets.

III. UPMEM Architecture Overview

UPMEM technology relies on the concept of processing in memory (PIM). Co-processing units called DPUs (standing for DRAM Processing Unit), are integrated into the main memory. They can execute operations directly into the DIMM, minimizing the data movement. This concept permits massive parallelization through the DPUs while the bandwidth is increased and the latency is reduced, compared to a standard parallel architecture. This makes this technology particularly efficient for data-intensive operations.

A. Architecture

DPUs are programmable co-processor optimized for data computing, designed to run small programs or routines on behalf of the CPU. They can be run simultaneously and are independent in term of code and data. Each of them is attached to a distinct bank of the memory with independent access so the bandwidth is improved since no bus is shared.

The DPUs are based on a RISC architecture with 24 32-bit registers per thread. The software implementation of an execution thread is called a tasklet and is based on hardware threads.

The DPUs have no data and instruction cache but two fast memories: the instruction RAM called WRAM and a scratchpad memory called MRAM, shared by all the tasklets. The global architecture is shown in Fig. 1. The MRAM must be explicitly managed by the programmer who can reserve areas for shared memories or for exclusive tasklet usage (heap allocation).

The architecture is designed to sit in the DRAM technology, which introduces several constraints. The power of the DPUs is limited compared to a standard processor: the frequency is around 750 MHz and operations they can perform are limited. For example they cannot perform floating point operations and the multiplication is expansive with an overhead of 30 cycles.

The strength of the technology lies in the data access. The DPUs support multi-threading up to 24 threads, in a way that the context is switched at every clock cycle. A thread executing a direct memory access is suspended until the transfer is complete, insuring minimum latency.

One DPU is attached to every 64 MBytes of memory. Hence, a 16 GBytes DIMM embeds 256 independent DPUs, and several of them can be added to a standard CPU. This makes a total of 4096 cores together that can support up to 24 threads, with 256 GBytes of memory. However, this massively parallel environment require an adaptation of the software.

B. Programming applications on UPMEM

On the programming level, two programs must be specified.

The host processor acts as a coordinator: it allocates the DPUs, loads the program into the DPUs, prepares the data, boots the DPUs and gathers the results.

The data-intensive part of the code is offloaded to the DPUs as tasklets. Several tasklets can run simultaneously on a DPU and synchronization primitives, as well as share memory mechanisms, are available to orchestrate the execution.

It can be seen as a distributed system at the server level. Note that the tasklet execution can be run asynchronously with the host program, allowing host tasks to be overlapped with DPUs tasks.

C. Communication between CPU and DPUs

The CPU has two mechanisms to communicate with the DPUs: through the MRAM or through a mailbox system.

Communication through the MRAM is similar to a standard communication between the CPU and the memory. The CPU stores some data into the MRAM, the DPU computes the data, store the result into the MRAM and the CPU loads the result. This mechanism is rather slows but allows the communication of large amounts of data.

The mailbox mechanism is faster but the amount of shared data does not exceed a few bytes, which makes it ideal for synchronization. The mailbox is located in the WRAM and can be accessed quickly by the DPU. Both the CPU and the DPU can perform post or receive operations on the mailbox and exchange data.

The DPUs cannot communicate directly but only through the host processor, which makes this operation vulnerable to bandwidth issues.

To sum up, UPMEM technology offers a massively parallel environment with increased bandwidth and minimum latency for data-intensive applications. The application has to be adjusted to consider the distributed architecture format. The main program supervises the data transfers and the tasklets execution while the DPUs handle the data-intensive part of the code.

IV. k-MEANS IMPLEMENTATION FOR THE UPMEM ARCHITECTURE

In this section, we explain how we made a distributed k-means. IV-A presents our algorithm of k-means using UPMEM architecture. We then give details about our implementation in IV-B.

A. *Highly Parallelized k-means Algorithm*

The main idea is to distribute the data across the DPU memories in order to offload the computation of the distances. The host processor communicates the centroids to the DPUs, gathers and merges the results. The main advantage is that the distances computations are delegated to the DPUs to be run in parallel near the data.

The following steps are performed:

1) The points are uniformly distributed between the DPUs.
2) The host processor chooses the initial centroids.
3) The host processor communicates the centroids to the DPUs.
4) The DPUs assign each point to the nearest centroid.
5) The DPUs start computing the new centroids with their partial information.
6) The host processor merges the partial results and computes the new centroids.
7) Repeat from 3 until the convergence criteria is met.
8) Return the clusters.

Those steps are illustrated in Fig. 2.

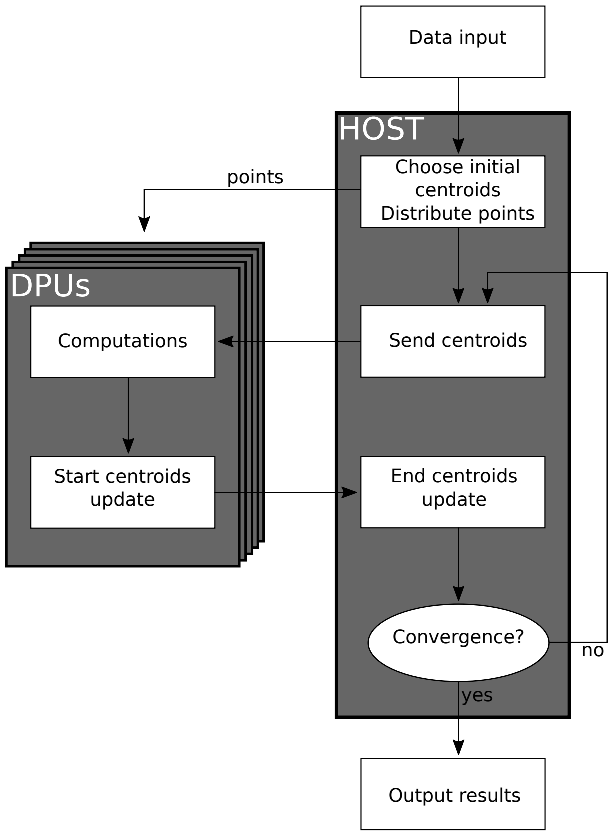


Fig. 2. **k-means on UPMEM.** The points are uniformly distributed across the DPUs. Computations consist in calculating the nearest cluster for each point.

B. *Implementation Details*

We expose now the details of the two aspects of the implementation: the host program, and the DPUs tasklet.

1) Host program: First, the dataset is processed to get the data and its characteristics (number of points and dimensions). Then we request the DPUs, and we initialize them by sending the tasklet program and general information (the value of k, or the number of threads requested by the user, ...). Next, the points are dispatched uniformly over the DPUs. The host enters the `while` loop and starts by sending the current centroids to all DPUs. It boots the DPUs and wait for them to have all completed their task. Then, the host retrieves partial centroids computed by the DPUs and performs a reduce task to get the new centroids for the next iteration. It also checks if the convergence criteria is met, by comparing the old and the new centroids. If this is not the case, this loop is repeated until convergence. Otherwise, the hosts writes the results in a logging file and terminates.

2) DPU tasklet: This program is run by every thread of each DPU. The MRAM stores the followings:

- global variables (e.g. the number of points),
- centroids,
- points, and
- new centroids.

First the thread retrieves the general informations, its ID, and the current centroids, and stores it into the WRAM. Then it loops over every point it is in charge of. It transfers the point from the MRAM to the WRAM. It performs some computation to determine the closest centroid and does a partial centroid update, by adding the contribution of the current point to the right centroid. Once this step is done for every point, the thread adds up its partial centroids computation to the global one in MRAM. We use a critical section as this memory access is shared between all threads.

3) Limits: This implementation could still be improved in many ways. A part of the data used in the tasklet is identical between threads. Therefore, shared memory could be added to store those data and avoid redundant computation.

Also, since the multiplication is rather expensive in this architecture, it could be interesting to use another distance without any multiplication during the first steps to approach the result, and only use euclidean distance for the final convergence.

V. EXPERIMENTAL EVALUATION

We now give details about our experimental setup in V-A, and present our results in V-B.

A. *Experimental Set Up*

As the architecture is not yet available, we use a cycle-accurate simulator that we run on a virtual machine to conduct the experiments.

Name	Number of points	Dimension	Value used for k
RANDOM-0	500000	34	3
RANDOM-1	1000000	10	5
RANDOM-2	100000	2	10

TABLE I
CHARACTERISTICS OF RANDOM DATASETS

Firstly, we will give some details about the datasets used to perform the experiments. Secondly, we will give an overview of some existing implementations of the k-means algorithm that we can use as a comparison. Finally, we will give the specifications of the virtual machine we used to conduct the experiments.

1) Datasets: We consider a set of n points with d coordinates. A dataset is thus a subset of $\mathbb{R}^{n\times d}$.

As the UPMEM hardware only accepts integer operations, we only consider datasets with integer coordinates. Experiments with floats could however be performed as further work, using a software library to emulate floating operations. We consider datasets with different characteristics in terms of number of points and dimension value. Experiments are performed with randomly generated datasets.

We generate random datasets using the following protocol:

1) We first compute k random points uniformly in an hypercube of edge of size 1024, these points will be used as centroid for clusters.
2) We add a point to the dataset by first choosing randomly one of the clusters, and then generating the point around it with a Gaussian distribution.
3) We thus obtain some float coordinates that we convert to integers.

This way, we obtain a dataset for which the result of the k-means algorithm is meaningful.

We chose different characteristics and seeds to select three random datasets, to use during evaluation. These are described in TABLE I.

2) Reference implementations: We aim to compare our program with some existing k-means implementations. Thus we give an overview of those we selected to perform this evaluation.

[14] gives a sequential k-means program written in C, as well as a parallel program using MPI. Message Passing Interface (MPI) is a standard library that allows to exchange messages between processes or nodes, and that is widely used for parallel computing.

[15] uses the Hadoop framework, which is an open source implementation of MapReduce, a powerful parallel programming technique. [16] provides a benchmark for k-means.

Finally, [17] gives a GPU implementation that uses the Cuda platform.

3) Virtual Machine specifications: The experiments are conducted on a virtual machine (QEMU Virtual CPU version (cpu64-rhel6), 4 cores 2.4 MHz, 4 GBytes RAM) configuration running Linux Fedora 20. Both the sequential and the simulator experiments are conducted on this architecture. Note that the simulator is taking some extra space into the main memory, thus we need to check that this doesn't affect the overall computation time. Firstly, we ensured that this does not induces extra swap accesses. Secondly, the time-consuming part is the tasklet computation which runs on the simulator thus it cannot be affected, and a little increase of the host computation time would not be significant in the overall computation time.

B. Experimental Results

1) Impact of multi-threading: In order to see the impact of the DPU threads, we run experiments with different number of threads, using the three random datasets previously presented. The runtime is obtained as follows: we measure the number of cycles taken for each DPU and iteration. We keep the value of the longest DPU, and we sum over the iterations. Finally, we compute the final runtime as we know DPU frequency is 750 MHz.

Results are given in Fig. 3. We can see the impact of the thread pipeline to ensure minimum latency while accessing memory. The runtime decreases as the number of threads increases, until reaching a plateau starting at around 8 threads. Therefore, we set the number of threads to 10 for all remaining experiments.

2) Impact of the number of DPUs: We now fix a dataset, we chose RANDOM-0, and measure runtime with different numbers of available DPUs. Results are given in Fig. 4.

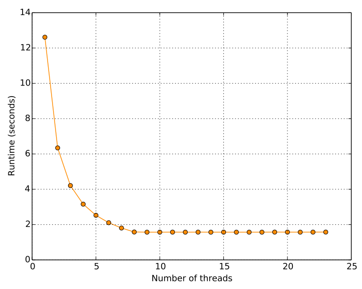
As we can see, the dependence is inversely proportional: if we multiply by two the number of DPUs, the runtime is divided by two. This result is consistent since the computation time per point is independent from the number of points the DPU has to process.

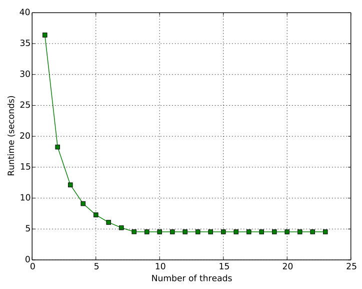
3) Comparison with Reference Implementation: We compare our implementation with the C sequential program presented in V-A2. We refer to it as SeqC, and to our implementation as X-DPUs, meaning the experiment was conducted using X DPUs. We consider random datasets previously presented. Runtime results are summarized in TABLE II. Because of the previous results, we can compute the number of DPUs required to be faster than the sequential version.

As we can see, having a low number of dimensions and centroids is the best case. This can be explained because a larger number of dimensions requires a larger amount of multiplications per point for the distance computation. A similar problem occurs with a larger number of centroids that requires a larger amount of computation per memory transfer [18].

VI. CONCLUSION

We first presented the concept of *Processing-in-Memory* and the data-intensive k-means algorithm which we chose to evaluate the UPMEM PIM architecture with. After introducing the architecture we explained how we implemented the k-means algorithm. We then evaluated our program and exposed some limitations due to the low computation power of the architecture. When some parameters for the k-means increase, computation becomes the bottleneck and thus the program


(a) RANDOM-0


(b) RANDOM-1

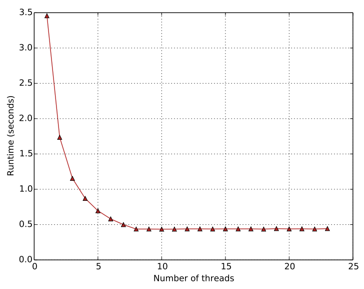

(c) RANDOM-2

Fig. 3. DPU runtime for different number of threads with random datasets

Dataset	RANDOM-0		RANDOM-1		RANDOM-2	
Algorithm	16-DPUs	SeqC	16-DPUs	SeqC	16-DPUs	SeqC
Runtime (s)	1.568	0.268	4.534	0.119	0.4353	0.0142
Faster than SeqC with	94 DPUs		610 DPUs		491 DPUs	

TABLE II
COMPARISON WITH C SEQUENTIAL PROGRAM ON RANDOM DATASETS

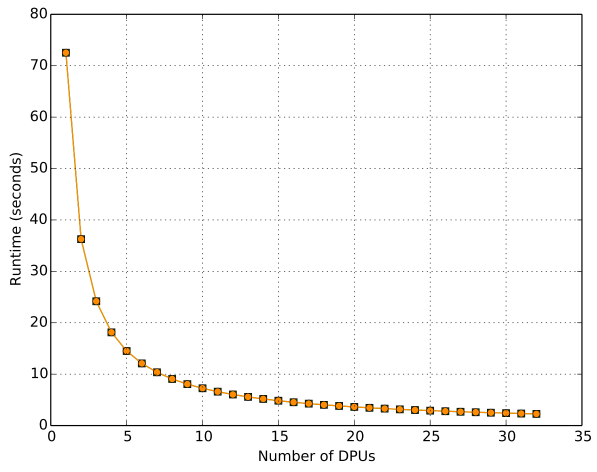

(a) RANDOM-1

Fig. 4. DPU runtime for different number of DPUs

becomes unsuitable for this architecture. In the end it is useful to find which level of memory bound makes a program running faster on UPMEM, because we already know that it works well with genomic text processing [19], [20].

We still need to compare with other parallel implementations to see if we have similar limitations or advantages. We also could conduct experiments with real datasets, for example from the UCI Machine Learning Repository [21] or Hadoop benchmarks [16] by adapting floating points to integers.

We could also go further on the hardware side. Having the actual physical device would allow us to evaluate it at large scale and study the impact of communications. Also, in a future version it might be possible that a hardware multiplier is implemented. As we have seen by profiling our program, 40% of the executed instructions are multiplication steps. Given that a multiplication takes about 30 cycles it would be a significant improvement and would remove the slow-down when the dimension is increased.

On the algorithmic side, a simple tweak would be to keep computed distances in DPU [10]. It would be interesting to study the trade-off between space and time. Another additions, harder to implement, is to keep only a border of points that can switch clusters [11]. This time, it is the possible involvement of the CPU in the computation that would be interesting to evaluate.

REFERENCES

[1] "UPMEM," http://www.upmem.com/.

[2] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[3] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, "Np-hardness of euclidean sum-of-squares clustering," *Mach. Learn.*, vol. 75, no. 2, pp. 245–248, May 2009. [Online]. Available: http://dx.doi.org/10.1007/s10994-009-5103-0

[4] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[5] R. Minnick, R. Short, J. Goldberg, H. Stone, and M. Green, "Cellular arrays for logic and storage." DTIC Document, Tech. Rep., 1966.

[6] P. Siegl, R. Buchty, and M. Berekovic, "Data-centric computing frontiers: A survey on processing-in-memory," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 295–308. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989087

[7] S. Guyetant, M. Giraud, L. LHours, S. Derrien, S. Rubini, D. Lavenier, and F. Raimbault, "Cluster of re-configurable nodes for scanning large genomic banks," *Parallel Computing*, vol. 31, no. 1, pp. 73–96, 2005.

[8] G. Georges, S. Derrien, S. Rubini, F. Raimbault, L. Amsaleg, and D. Lavenier, "Remix: une architecture pour la recherche dans les masses de données indexées," in *Symposium en archichecture de machines (Sympa) 2006*, 2006, pp. 142–153.

[9] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[10] A. M. Fahim, A. M. Salem, F. A. Torkey, and M. A. Ramadan, “An efficient enhanced k-means clustering algorithm,” *Journal of Zhejiang University-SCIENCE A*, vol. 7, no. 10, pp. 1626–1633, 2006.

[11] C. M. Poteraş, M. C. Mihăescu, and M. Mocanu, “An optimized version of the k-means clustering algorithm,” in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. IEEE, 2014, pp. 695–699.

[12] W. Zhao, H. Ma, and Q. He, *Parallel K-Means Clustering Based on MapReduce*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 674–679.

[13] J. Zhang, G. Wu, X. Hu, S. Li, and S. Hao, “A parallel k-means clustering algorithm with mpi,” in *Proceedings of the 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, ser. PAAP ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 60–64.

[14] W. keng Liao, “Parallel k-means data clustering for large data sets,” 2013. [Online]. Available: http://users.eecs.northwestern.edu/w̃kliao/Kmeans/kmeans_int64.html

[15] E. Gabriel, “Hadoop k-means data clustering.” [Online]. Available: http://www2.cs.uh.edu/g̃abriel/courses/cosc6339_s15/kmeans.java

[16] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, “Puma: Purdue mapreduce benchmarks suite,” 2012.

[17] S. Giuroiu, “Cuda k-means clustering,” 2012. [Online]. Available: http://serban.org/software/kmeans/

[18] M. A. Bender, J. Berry, S. D. Hammond, B. Moore, B. Moseley, and C. A. Phillips, “k-means clustering on two-level memory systems,” in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS ’15. New York, NY, USA: ACM, 2015, pp. 197–205. [Online]. Available: http://doi.acm.org/10.1145/2818950.2818977

[19] D. Lavenier, C. Deltel, D. Furodet, and J.-F. Roy, “BLAST on UPMEM,” INRIA Rennes - Bretagne Atlantique, Research Report RR-8878, Mar. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01294345

[20] ——, “MAPPING on UPMEM,” INRIA, Research Report RR-8923, Jun. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01327511

[21] M. Lichman, “UCI machine learning repository,” 2013. [Online]. Available: http://archive.ics.uci.edu/ml