**MUCH-SWIFT: A High-Throughput Multi-Core HW/SW Co-design K-means Clustering Architecture**

Hadi Mardani Kamali  
Department of Electrical and Computer Engineering  
hmardani@gmu.edu

Avesta Sasan  
Department of Electrical and Computer Engineering  
asasan@gmu.edu

**ABSTRACT**

K-mean clustering is an essential tool for many big data applications including data mining, predictive analysis, forecasting studies, and machine learning. However, due to large size (volume) of Big-Data, and large dimensionality of its data points, even the application of a simple k-mean clustering may become extremely time and resource demanding. In this paper, we propose a two-level filtering algorithm based on binary kd-tree structure, which considerably decreases the time of convergence in K-means algorithm for large datasets. The proposed modification to the classification algorithm, evolves the SW to naturally divide the classification into smaller data sets, based on the number of available cores and size of logic available in an FPGA. The empirical results show that on a multi-core FPGA, provides 330× speed-up compared to a conventional SW-only solution.

**KEYWORDS**

Clustering, FPGA, K-means, Filtering, kd-tree

1 INTRODUCTION

Many current applications of cloud computing, especially when combined with the Internet of Things, are of predictive, data mining, machine learning, or forecasting nature [1, 2, 6]. Such applications are in need of categorizing and clustering very large and high-dimensional input data-sets as a part of their larger computational flow. The ability to classify and cluster a desired set of data is an essential part of building knowledge from data. However, as the size and dimensionality of data increases, the execution time of clustering algorithms grows superlinearly, making it a big challenge when dealing with Big-Data.

The goal of clustering is to classify the data according to a specific metric such that objects within a cluster/group, in terms of having a feature are similar. A clustering algorithm may be supervised (hierarchical) or unsupervised (un-nested), exclusive or fuzzy, and could be complete or partial. Depending on the type of clustering algorithms used, the resulting clusters may be well separated, prototype-based (centroid-based), graph-based, or density-based [4].

K-means is one of the simplest and yet most used [4, 5] centroid-based unsupervised clustering algorithms. However, its applicability to Big-Data depends on the scalability of its software (SW) implementation with respect to the available hardware (HW). One of the most promising HW platforms that is leveraged for achieving considerable speedup in big data applications is FPGAs. FPGA solutions enable higher parallelism than clusters of CPUs or GPUs at a much lower cost, at the expense of greater mapping overhead. In order to improve the usability of FPGA solutions when dealing with semi-parallel applications, the FPGAs are equipped with mid to high-performance multi-core processors (e.g. ARM Cortex A9, A12, A15). The existence of multiple mid to high-performance cores on the same die as FPGA, as well as hundreds of thousands of fine-grained logics and coarse-grained communications within FPGA [18, 19], improves the efficiency of HW/SW co-design and provides greater flexibility and wider applicability for use of FPGAs accelerated solutions.

The execution time of k-clustering algorithms could be improved by means of both SW and HW. For example, on the SW side, one could use (1) binary kd-tree structure for dividing search space members into “boxes” [7], and (2) triangle inequality for avoiding redundant distant calculation [8]. On the HW side, more capable or additional computing resources reduce the computational time. In this paper, we present MUCH-SWIFT: a HW/SW partitioning architecture that considerably reduces the execution time of kd-tree clustering algorithm compared to the state of the art FPGA mapped solutions. For this purpose, we have implemented a mapping and an aggregation function on top of the kd-tree clustering algorithm. This approach allows us to divide the work across logic and multiple cores in an FPGA to gain the maximum achievable speedup by utilizing all available resources. Additionally, we implemented a custom high-performance DMA in order to transmit data between host and FPGA via PCI Express (PCIe) interface, which considerably reduces the execution time overhead related to data transfer time, and helps us achieve considerable speedup in comparison with a conventional software-based solution.

The rest of the paper is organized as follows. The k-means theory and algorithm are described in Section 2. Section 3 briefly illustrates the structure of binary kd-tree for filtering algorithm. The architecture of MUCH-SWIFT and its parallel structure is elaborated in Section 4. Experimental results are shown in Section 5. Section 6 covers the related work. And finally, Section 7 concludes the paper.

2 K-MEANS CLUSTERING ALGORITHM

K-means is one of the simplest and fastest partitioning algorithms which is widely used for unsupervised centroid-based clustering. It divides the input dataset into "k" groups, called clusters, where all members in a cluster are similar. The k-means clustering is a centroid-based algorithm meaning each cluster has a prototype which is the indicator of that cluster. Hence, during the clustering, each data point is assigned to a cluster whose centroid is the closest. In k-means clustering algorithms, in order to calculate the distance between each point and the cluster centroids, three conventional distance metrics have been used: Manhattan, Max, and Euclidean [6]. For instance, if we suppose that each data point is a vector \( \vec{dp} = (p_1, p_2, \ldots, p_m) \), the Euclidean distance can be defined as follow:

\[
\text{EuclidDist}(\vec{dp}, \text{cent}) = \left( \sum_{i=1}^{m} (dp_i - \text{cent}_i)^2 \right)^{\frac{1}{2}}
\]  

(1)

The k-means algorithm first initiates \( k \) centroids. Then it enters an iterative process where each iteration consists of two steps: (1)
Algorithm 1: Filtering algorithm using binary kd-Tree [7]

1: function Filter(kdNode u, CandidateSet Z)
2:     C ← u.cell;
3:     if u is a leaf then
4:         z* ← the closest point in Z to u.point;
5:         z*.WgtCent ← z*.WgtCent + u.point;
6:         z*.count ← z*.count + 1;
7:     else
8:         z* ← the closest point in Z to C’s midpoint;
9:         for all z ∈ Z \ {z*} do
10:             if z.isFather(z*, C) then
11:                 Z ← Z \ {z*};
12:             if |Z| <= 1 then
13:                 z*.WgtCent ← z*.WgtCent + u.wgtCent;
14:                 z*.count ← z*.count + u.count;
15:         end if
16:     end if
17:     Filter(u.left, Z);
18:     Filter(u.right, Z);
19: end function

Alg. 1 depicts the kd-tree filtering algorithm. In each node, Z determines a subset of candidate centroids. If we suppose that we have k clusters, the candidate centroids for the root node are all k centroids, and the candidates for each internal node are a subset of k clusters. Additionally, the candidates of each node are the nearest neighbors for some points. For each node, the distance between z* ∈ Z and the midpoint of cell is calculated and compared with z ∈ Z \ z*. Then, the sub-tree at the larger distance is pruned.

4 MUCH-SWIFT ARCHITECTURE

MUCH-SWIFT is a HW/SW co-design architecture for accelerating the k-means algorithm. It benefits from utilizing multiple hard-core processors and plenty of resources in a single FPGA that are available in the same chip. For the implementation of MUCH-SWIFT, we have utilized the ZYNQ Ultrascale FPGA board. The ZYNQ Ultrascale+ architecture has four Cortex-A53 with maximum clock frequency of 1.5 GHz, and two Cortex-R5 with max frequency of 600 MHz. It is equipped with 1 GB DDR3 off-chip DDR3 memory, and ZU9EG FPGA chip with around 600K logic cells. In our implementation, the MUCH-SWIFT SW is distributed on Processor System (PS) (the Quad Cortex-A53 processor and the dual Cortex-R5 coprocessor). On the HW side, the parallel arithmetic cores that are required for Manhattan distance calculations, comparators, and updaters are mapped to the Programmable Logic (PL) on ZU9EG. Fig. 1 illustrates the overall architecture of MUCH-SWIFT in ZCU102. Our proposed two-level algorithm maximizes the utilization of cores; each Cortex-A53 core is responsible for evaluating and analyzing one quarter of data points independently. Additionally, in order to reduce the search time, we employ a binary kd-tree structure to filter (prune) some nodes and their children, which their candidates are not the nearest centroid. Then, in order to maximize the utilization of all four Cortex-A53 cores, we build N parallel clusters (N being the number of available cores, which is 4 in our experimental board) by dividing the original data-set into N smaller data-sets at the top of the kd-tree. After clustering the sub-datasets, they are merged together and the filtering algorithm is invoked on top of the merged clusters. Using this two-layer clustering approach increases convergence, which decreases the required number of iterations for the clustering. Furthermore, MUCH-SWIFT is able to process large datasets by using a DMA-based PCIe interface and DDR3 memory in ZCU102 without any significant throughput degradation. In our implementation, one of Cortex-R5 is made responsible for handling custom DMA access between PCIe and DDR3 memory, while another Cortex-R5 generates the initial states of each quarter of data points as well as initial values of centroids. In addition, the second Cortex-R5 is made responsible for controlling the update procedure after pruning in kd-tree structures.

4.1 Parallelism in kd-tree Traversal

To speed up the kd-tree clustering process, we implement a two-layer clustering mechanism. For the first level of clustering, we randomly divide the input data into four independent subsets, and each Cortex-A53 core is made responsible for processing a quarter of data as an independent kd-tree structure into k clusters. From this process, we obtain 4k centroids, and a cluster associated with each centroid. Then we combine the clustering result of four divided subsets into one. To achieve this, we combine a cluster in each subset with its three nearest clusters in other subsets. The final step in this process is to run the second level of clustering to update the centroids and cluster members. Considering that we are working
on very large datasets and the initial assignment of data to the four subsets was random, the clustering results of the 4 subsets are expected to be very close to one another. Hence, the merged centroids and their associated members are expected to be very close to the final result. In the result, it is expected that the number of iterations for the second level of k-clustering to be very small.

**Algorithm 2 Two-level k-clustering Algorithm by Using 4 parallel Binary kd-Tree Structures**

1: function ParallelClustering(DataPoint_Set DP)
2:     for i = 0 to 3 do
3:         DataPoint_Set QDP[i] ← Quarter(DP, i);
4:         kdNode kdu[i] ← Gen_KdTree(QDP[i]);
5:         CandidateSet Z, Update[i] ← Lloyd(QDP[i]);
6:         CandidateSet Z, Current[i] ← Z, Update[i];
7:     for i = 0 to 3 do
8:         Filter(kdu[i], Z, Update[i]);
9:         while Z, Update[i] ≠ Z, Current[i] do
10:            Z, Current[i] ← Z, Update[i];
11:       Filter(kdu[i], Z, Update[i]);
12:      kdNode kdu, top ← Combine(kdu[0 : 3]);
13:     while Z is updated do
14:         Filter(kdu, top, Z);
15:   end function

Alg. 2 illustrates the pseudo-code of our proposed two-layer partitioning flow. During the initialization state, the dataset is divided into four separate subsets via `Quarter` function. Then, a kd-tree is generated for each sub-dataset, and the Lloyd function is employed for choosing initial centroids [3]. Then, parallel tree traversal is accomplished (Lines 8–14), where each Cortex-A53 core is made responsible for transceiving data to/from PL in order to calculate and update its corresponding kd-tree characteristics (i.e., centroids and clusters) in parallel.

### 4.2 No Limit for Dataset Size via High Throughput DDR3 Memory

We employ DDR3 off-chip memory in ZYNQ Ultrascale+ to maximize the feasible size of data. ZYNQ Ultrascale+ provides an efficient and fast DDR3 memory, which is accessible from both PS and PL, illustrated in Fig. 1. The capacity of this memory is 1 GB, and it has a 128 bits data-bus for read/write access. Also, as it can be seen in Fig. 1), we implement a BRAM-based bridge (BRAM-based FIFO) between DDR3 and PL in order to transfer data from PL to DDR3 and vice versa. In order to minimize the required BRAM-based bridge size between DDR3 and PL, we evaluate the data size for each level of tree traversal separately. Similar to [13], hierarchical access allows us to release and reuse the memory at each level (depth) before starting the next level (depth) of tree. In addition, all data is permanently kept in DDR3; hence, overwriting the data can be accomplished without any throughput degradation. As a result, the large size of DDR3 enables us to maximize the dataset size as required. For instance, suppose that MUCH-SWIFT is configured to classify \( N = 100000 \) data into \( K = 1024 \) clusters. In the worst case, the structure of kd-tree is like a degenerate tree. In this case, we need \( (N - 1) \times (K) \times (\log K) \approx 122 \) MB, which is much less than the DDR3 memory, i.e., 1 GB.

### 5 IMPLEMENTATION RESULTS

We targeted a large Xilinx ZYNQ-based SoC architecture (ZCU102 evaluation board) to evaluate our proposed architecture. The PS consists of a quad Cortex-A53 core and a dual Cortex-R5 core. A53 cores are made responsible for transceiving data to/from each core.
Table 1 reports MUCH-SWIFT’s resource utilization with the different number of clusters. When increasing the number of clusters, we need more resources for parallelism, and the available resources on FPGA are limited. So, there is a limit for the number of clusters on FPGA for fully parallel architecture. As reported in Table 1, the maximum number of clusters (for fully parallel architecture) is 20, and for the applications with more clusters, we have to share the parallel modules between clusters uniformly. Note that, 20 clusters means that we can implement $20 \times 4 = 80$ parallel modules on ZU9EG, which is significantly large. Also, our implementation employs the highest proportion of BRAMs and DSPs in order to maximize the number of parallel arithmetic cores.

### 6 RELATED WORK

A HW/SW co-design architecture is implemented in [9] based on NIOS 1.1. But, the HW/SW interface is Peripheral Bus Module (PBM), whose serial infrastructure considerably limits the throughput. Unlike a HW/SW architecture, pure FPGA-based designs [10, 11] provide significant speed-up against HW/SW co-designs by using fixed-point arithmetic. But, on-chip FPGA memories (like BRAMs) is a big restriction for storing large datasets.

In order to avoid redundant distance calculations, triangle inequality [8] has been implemented successfully in [12]. However, the size of data points is truncated to 8 bits, which is small. A filtering algorithm by using a kd-tree structure is implemented in [13]. Due to using on-chip memories for storing data, it is only able to store 64K data simultaneously. Also, the size of data points is limited to 16, and all computations are based on fixed-point arithmetic.

Another pure FPGA-based k-means clustering architecture is implemented in [14]. The number of clusters is fixed in this architecture, and changing it needs re-synthesis and re-implementation. A computer cluster which consists of multiple FPGA-CPU pairs is implemented in [15]. In this architecture, map-reduce programming is designed to allow easy scaling and parallelization across the distributed computer system. Although this evaluation on multiple FPGA-CPU pairs shows considerable throughput against baseline software implementation, it needs evaluating the case for utilizing multiple FPGAs in processing larger datasets.

A specific FPGA accelerator for the Intel QuickAssist FPGA platform is implemented in [16], which provides an integration between threads in a CPU and an Accelerated Function Unit (AFU) in QuickAssist FPGA. But, it is applicable only for this specific type of FPGA platform, i.e. IntelAssist.

### 7 CONCLUSION

In this paper, we proposed MUCH-SWIFT, an FPGA-based architecture for parallelization of the k-clustering algorithm and a modified two-layer filtering optimization. The MUCH-SWIFT employs all processing cores in the ZYNQ Ultrascle+ SoC to reduce the computation time and a two-layer filtering algorithm designed for parallel processing of binary kd-tree structures. Furthermore, by employing ZYNQ Ultrascle+ and utilizing its DDR3 memory, MUCH-SWIFT increases the feasible size of its input datasets. Additionally, MUCH-SWIFT benefits from our proposed HW/SW co-design architecture, which provides a high-throughput DMA-based PCIe channel for transceiving datasets between the host and ZYNQ SoC. By using this HW/SW co-design architecture, the MUCH-SWIFT achieves around 330× speedup compared to a software-only solution.

### REFERENCES


