A Flexible K-Means Operator for Hybrid Databases

Zhenhao He David Sidler Zsolt István Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zürich, Switzerland
zhe@student.ethz.ch david.sidler@inf.ethz.ch zsolt.istvan@inf.ethz.ch gustavo.alonso@inf.ethz.ch

***Abstract*—The K-means algorithm is widely used in unsupervised learning and data exploration. It is less used in analytical databases due to its high computational cost. K-means has been explored in great detail, mostly focusing on performance. However, in emerging hybrid CPU-FPGA databases where memory bandwidth is shared across software and hardware operators, two additional requirements arise. One is parameterization to avoid frequent reprogramming. The other is concurrent use to balance memory bandwidth and computation. Our design supports two operational modes that can be chosen at runtime, one for high query throughput and one for evaluating multiple clusters concurrently. The former targets speed up, while the latter targets efficient bandwidth utilization by increasing the amount of computation per input byte. Our design is competitive when compared to both existing FPGA-based solutions as well as highly optimized multi-core software implementations.**

I. Introduction

K-means clustering is used as a building block for unsupervised learning tasks spanning several application domains. Due to its computational intensity it is a common target for software and hardware optimizations [1]–[3]. In this work, we extend the state of the art by providing flexible runtime parametrization of the algorithm while maintaining a throughput comparable to the fastest software and hardware solutions.

The need for exploring flexible circuits that can adapt to different input parameters (number of dimensions and centroids) arises from the adoption of hybrid CPU-FPGA architectures where the overhead of offloading is lower than in PCIe-attached setups, opening up opportunities for FPGA acceleration while making reconfiguration less appealing. In this paper we focus on extending databases [4] running atop hybrid architectures with hardware-accelerated analytical operators [5], [6] where workloads change often and comprise many concurrent queries working on varying amounts of data, motivating the need for parameterizable designs. Further, in hybrid architectures, memory bandwidth is shared between operators concurrently running on the CPU and FPGA. To ensure that hardware operators do not prevent others from making progress, their design should allow to adjust the required memory bandwidth. Instead of slowing down processing when memory bandwidth is scarce, the design should offer more computation on the same data: one example is running the clustering algorithm with different parameters concurrently.

Running the algorithm with different parameters addresses a key challenge of clustering operators, namely determining the "right" number of clusters. The most common method to detect the value of k is called the "elbow method". This traditionally requires multiple runs over the same dataset using a different numbers of clusters until adding another cluster does not significantly decrease the squared sum of errors within the clusters. On an FPGA, this computation can be efficiently parallelized, resulting in predictable runtime.

The contributions of this paper are as follows:

- A runtime parameterizable design that can accommodate different number of dimensions and centroids while offering the same constant processing rate thanks to its pipelined architecture.
- A complete FPGA implementation of the K-means algorithm including assignment, update, and error calculation.
- Two operational modes: *(1)* throughput oriented using the on-chip parallelism to accelerate iterations, and *(2)* bandwidth-conserving to compute multiple centroid numbers in parallel, leading to a faster exploration of clusters.

II. K-means Algorithm

K-means is an unsupervised clustering algorithm that groups data points into a predefined number of clusters k. The algorithm consists of three steps: (1) the initialization step, which picks k random centroids, (2) the assignment step, where each point is assigned to its closest centroid, and (3) the update step, where the centroids are recalculated as the mean of the points assigned to them. To determine the distance between data points and centroids, we use the Euclidean distance. The algorithm is either executed for a fixed number of iterations or until the centroid assignments no longer change. The number of centroids k is an input to the algorithm, however it is difficult to choose the most suitable value of k without prior information of the dataset. In order to find the optimal number of clusters, the algorithm is run on a range of k values until convergence. The sum squared error (SSE) as a function of k is plotted and the "elbow point", where the rate of decrease sharply shifts, can be used as a guide to determine k.

III. Design Overview

The K-means operator consists of three main parts: (1) the *Controller* which requests data from the main memory and writes back the results, (2) the *Assignment* pipeline which calculates the distance between a data point and all the centroids and assigns it to the closet centroid, and (3) the *Update* module which updates the centroids in-between iterations. The operator processes 32 bit fixed point values.

The parallel pipelines deployed on the FPGA can be utilized in two different modes: *High Bandwidth* and *Elbow*, both utilizing all deployed pipelines(Figure 1). In the *High Bandwidth* mode, they can all be assigned and parametrized with the same

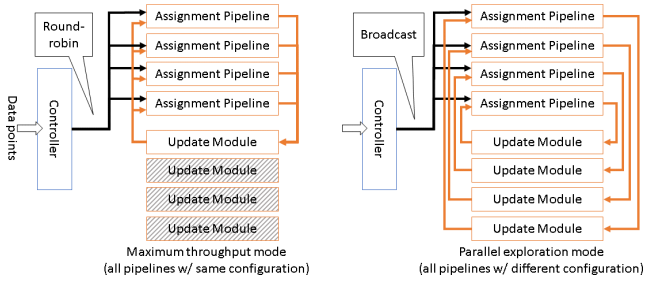


Fig. 1: The operator can be parametrized at runtime for either high throughput meaning all pipelines execute the same configuration or for parallel exploration where each pipeline has possibly a different configuration.

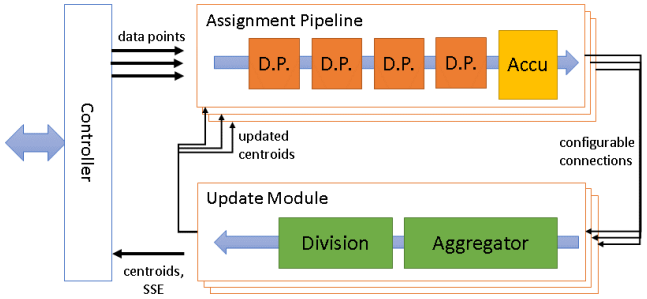


Fig. 2: Architecture of the operator with multiple *Assignment* pipelines and *Update* modules which can be dynamically interconnected at runtime.

number of clusters k providing the highest bandwidth for a specific k and maximizing memory bandwidth usage. In the *Elbow* mode, the user can partition and assign the pipelines to concurrently run different numbers of k for faster exploration of the space reducing the memory bandwidth usage.

In the former mode only one *Update* module is enabled since all pipelines work together and their results have to be aggregated after each iteration. In contrast, in the latter mode, pipelines work independently in parallel, each assigned to a different *Update* module. Apart from these two configurations, pipelines can be tiled in any power-of-two configuration. For instance, in our evaluation we use two pipelines per cluster number, calculating 8 different configurations on 16 pipelines.

A. Controller

When the operator starts, the *Controller* reads the initial centroids from the main memory. The input data is in a row format meaning all dimensions of a data point are contiguous in memory. After the distance processors in the *Assignment* pipeline are initialized with these centroids, the *Controller* starts fetching data points from memory and, depending on the mode, either broadcasts or distributes them in round robin fashion to the pipelines. After the algorithm is executed for the intended amount of iterations, the *Controller* writes the final centroids and the corresponding sum squared error (SSE) back to the main memory where it can be read by the application.

B. Assignment Pipeline

The *Assignment Pipeline* consists of an array of distance processors calculating the distances between each data point and all centroids. As shown in Figure 2, the data points are streamed through the array one dimension at a time. Each processor calculates the Euclidean distance between its stored centroid and the data points streaming through. This distance is compared with the minimum distance calculated from the previous processors and the minimum of these two values together with the corresponding cluster assignment is forwarded. In this way, the minimum distance and the corresponding cluster assignment can be obtained at the end of the array when the last dimension of a data point is output. The output of the distance processor array is fed into the Accumulator (*Accu*) module which accumulates the data points per cluster and maintains counters of how many data points are assigned to each cluster. After all data points are processed by the array, the accumulator pushes its results to the *Update* module.

C. Update Module

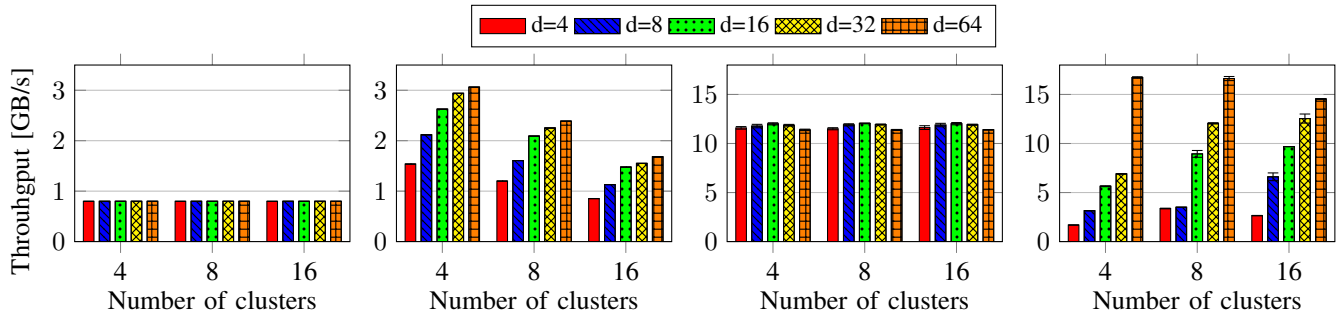
The *Update* module contains an *Aggregator* and a *Divider*. The *Aggregator* aggregates the data points, counts of assignments, and the SSE from multiple pipelines. The aggregated values are forwarded to the *Divider* which calculates the new centroids. The *Update* module then pushes the new centroids to the corresponding pipelines to update the centroids stored in the distance processors. The connections between pipelines and *Update* modules are configured at runtime and depend on the operational mode (Figure 1). After the last iteration, the final centroids and SSE are forwarded to the *Controller* which writes them to the main memory.

D. Run-time parametrization

Both the number of dimensions per data point and the number of clusters can be parametrized at runtime. The distance processors receive with each dimension a flag indicating if this is the last dimension of the data point. This flag is set by the *Controller* and therefore, the distance processors themselves are oblivious to the number of dimensions. Our operator is deployed with a fixed number of distance processors per pipeline, however the number of clusters can be parametrized at runtime to a lower number by disabling the unused distance processors. A disabled processor passes data along but does neither calculate the distance nor update the assignment.

E. Software Integration

For integration with the software we use the open-source framework Centaur [7], which provides software abstractions for the hardware operators and a thin hardware layer to facilitate concurrent and dynamic offloading of multiple operators to the FPGA. When the software calls the hardware operator through Centaur, it provides the memory pointers, runtime parameters and the operational mode. When an operator terminates, a *done* signal is send to Centaur which then notifies the user application that the operator terminated.



(a) FPGA using a single pipeline (b) CPU using a single thread (c) FPGA using 16 pipelines (d) CPU using 14 threads

Fig. 3: Throughput using synthetic data, with a varied number of clusters and dimensions.

IV. Evaluation

A. Setup

The presented K-means operator is evaluated on the second generation Intel Xeon+FPGA machine[1]equipped with a 14-core Xeon Broadwell E5 clocked at 2.4 GHz, 64 GB main memory, and an Arria 10 FPGA connected over 1 QPI and 2 PCIe links to the CPU's memory controller resulting in up to 17 GB/s read bandwidth. The operator is deployed with 16 pipelines running at 200 MHz supporting up to 16 clusters and 64 dimensions. For comparison we use a highly optimized multi-core K-means software implementation [1] taking advantage of SIMD and MIMD parallelism and minimizing data transfers between registers, cache, and main memory.

A real world Iris dataset[2]and a synthetic one are used. The former consists of 150 data points describing 3 types of flowers with 4 dimensions. We expanded it by adding uniform noise within 10% of the original values. The latter one has up to 64 dimensions following Gaussian multi-variate distributions.

B. Flexibility

We evaluate flexibility on the synthetic dataset by varying the number of dimensions and clusters. The performance of the FPGA with one and 16 pipelines and the software with one and 14 threads is measured. Figure 3 shows a stable throughput independently of the choice of dimensions and clusters. In both cases the throughput by the FPGA implementation is close to its theoretical maximum of 0.8 GB/s for single pipeline (32 bits per cycle at 200 MHz) and 12.8 GB/s for 16 pipelines. While a software based solution can provide the same amount of flexibility, its performance varies depending on the input parameters and is less predicable. Figure 3b shows that the throughput of a single thread decreases with increasing number of clusters. However in the case of 14 threads, the software implementation benefits from more clusters and dimensions, since this allows better parallelization of the work among multiple threads (Figure 3d). At the same time for a low number of dimensions and clusters increasing the number of threads only shows a marginal benefit.

[1]Results in this paper were generated using pre-production hardware and software, and may not reflect the performance of production or future systems.

[2]https://www.kaggle.com/uciml/Iris/data

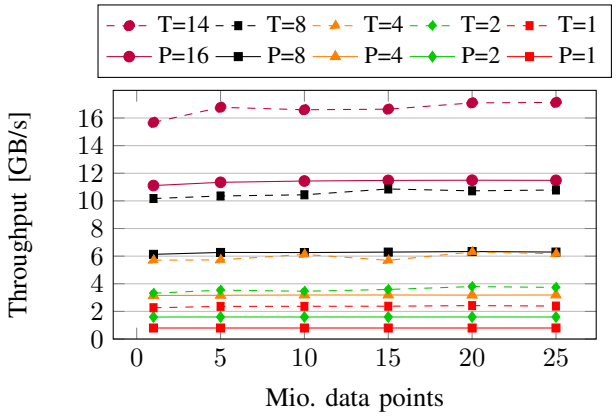


Fig. 4: Throughput for the synthetic dataset with 64 dimensions and k set to 8 comparing CPU and FPGA while increasing the number of pipelines (P) and threads (T).

C. Throughput

The throughput of our operator is evaluated in the *High bandwidth* mode, meaning all active pipelines are configured with the same centroids and number of clusters. As seen in Figure 4, the throughput of the circuit increases linearly with the number of parallel pipelines used. With 16 pipelines our design reaches 11.4 GB/s close to the theoretical peak of 12.8 GB/s. As a comparison, we run the multi-core K-means software implementation with increasing number of threads. As can be seen from the previous experiment, the chosen configuration(k = 8, d = 64) matches the software implementation. The FPGA peak performance is slightly higher than the 8-threaded execution. By using 14 threads the CPU benefits from the high parallelism and memory bandwidth and reaches up to 16.7 GB/s. From these results, we conclude that the FPGA can match at least 10 cores in terms of absolute performance while providing fully predictable response times for all parameter ranges.

D. Exploration: Elbow mode

The Elbow mode allows for a faster exploration of the optimum k while reducing the overall required memory bandwidth. Figure 5 compares the concurrent evaluation of 8 clus-

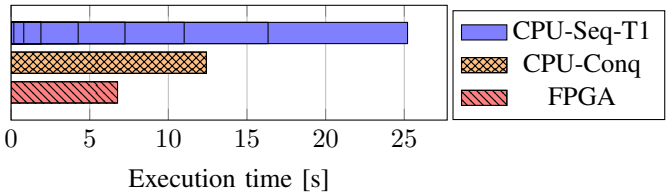


Fig. 5: Evaluation of multiple k, both sequentially and concurrently on software and concurrently in hardware

Resources	ALM		BRAM		DSP	
Controller	779	0.18%	0	0%	1	0.06%
1x Assignment	8212	1.92%	35	1.29%	55	3.62%
1x Update	3768	0.88%	19	0.70	0	0%
Total for P=1	13242	3.10 %	54	1.99%	56	3.69%
Total for P=8	86614	20.27%	432	15.92%	441	29.05%
Total for P=16	171550	40.17%	856	31.55%	881	58.04%
Centaur	87047	20.38%	429	15.81%	0	0%

TABLE I: Resource consumption of full operator including Centaur with single clustering module and 1, 8, 16 pipelines

ters using 2 pipelines each on the FPGA to a single threaded execution of the software implementation which evaluates 8 different clusters sequentially. The concurrent evaluation of multiple k's shows a clear performance benefit. To take the multi-core parallelism of the CPU into account, we also run all 8 clusters concurrently on the CPU, each assigned to a a different core (single-threaded). This reduces the execution time over a single thread but remains inferior to the FPGA.

To verify correctness of the FPGA operator, we run the elbow method on the expanded Iris dataset and compared it to results from a software implementation using either fixed point or floating point arithmetic. All three results are consistent and lead to an optimal cluster number of 3 given the 3 flower types.

E. Resource Usage

Table I shows the resource usage for the major components. Both the *Assignment* pipeline and the *Update* module have a very low resource usage allowing the desing to scale out in the number of pipelines and thereby increasing the bandwidth. The resource usage from 1 to 16 pipelines increases linearly using up to 40% logic resources. DSPs are mainly used for the distance calculation in the *Assignment* pipeline.

V. Related Work

There is a wide range of research on accelerating K-means using FPGAs [8], [9]. There are mainly two architectures, one based on a systolic-array of distance processors [10], [11] and one in which all distances are calculated in parallel followed by a reduction tree to find the minimum distance [12], [13]. We opted for the former, since it allows for pipeline parallelism and scales better to maximize bandwidth and DSP utilization. In the latter one, resources for the tree increase with the number of distance processors limiting scalability. Most existing work either processes floating or fixed point values. Estlick et al. [12] showed that using fixed point instead of floating point for the distance calculation has a negligible impact on the final accuracy. Many existing work offloads only parts of the algorithm such as the assignment [10], [12]. The benefit of offloading the entire algorithm due to reduced communication with the CPU, was shown by [11]. In addition to implementing the entire algorithm in the FPGA, our design provides runtime parametrization in terms of number of clusters and dimensions.

Several recent efforts use OpenCL [2], [3] to accelerate K-means clustering on FPGAs: [2] reports up to 11 GB/s of throughput which is similar to ours but neglecting the data transfer between host and device and only providing limited flexibility in the number of clusters, while the number of dimensions is fixed. Further, they target an accelerator deployment, whereas our design targets hybrid architectures. Hussain et al. [13] present a K-means implementation that achieves flexibility through partial reconfiguration allowing to change the distance function or run multiple K-means kernels on different datasets in parallel.

In software, the algorithm is compute-bound for most parameter combinations, therefore techniques such as vectorization [1] lead to significant speedups.

VI. Conclusion

The presented K-means implementation features flexibility in terms of runtime parameterization and memory bandwidth usage while having comparable performance to an optimized multi-core CPU implementation and out-performing existing FPGA solutions. The trade-off between memory bandwidth usage and performance can be adjusted through the assignment of pipelines to a single or multiple K-means configurations.

Acknowledgment

We would like to thank Intel for their generous donation of the Xeon+FPGA prototypes. This work is partially funded by Microsoft through the Swiss Joint Research Center.

References

[1] C. Böhm, M. Perdacher, *et al.*, "Multi-core K-means," in *SIAM'17*.
[2] Q. Y. Tang and M. A. Khalid, "Acceleration of K-means algorithm using Altera SDK for OpenCL," *ACM TRETS*, vol. 10, no. 1, p. 6, 2016.
[3] Z. Wang *et al.*, "Melia: A MapReduce framework on OpenCL-based FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, 2016.
[4] D. Sidler, Z. István, *et al.*, "doppioDB: A hardware accelerated database," in *SIGMOD'17*.
[5] K. Kara, D. Alistarh, *et al.*, "FPGA-accelerated dense linear machine learning: A precision-convergence trade-off," in *FCCM'17*.
[6] M. Owaida and G. Alonso, "Application partitioning on FPGA clusters: Inference over decision tree ensembles," in *FPL'18*.
[7] M. Owaida, D. Sidler, *et al.*, "Centaur: A framework for hybrid CPU-FPGA databases," in *FCCM'17*.
[8] T. Saegusa and T. Maruyama, "An FPGA implementation of K-means clustering for color images based on Kd-tree," in *FPL'06*.
[9] X. Wang and M. Leeser, "K-Means clustering for multispectral images using floating-point divide," in *FCCM'07*.
[10] M. Gokhale, J. Frigo, *et al.*, "Experience with a hybrid processor: K-Means clustering," *The Journal of Supercomputing*, 2003.
[11] W.-C. Liu, J.-L. Huang, *et al.*, "Kacu: K-means with hardware centroid-updating," in *Conference, Emerging Information Technology 2005.*
[12] M. Estlick *et al.*, "Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware," in *FPGA'01*.
[13] H. M. Hussain, K. Benkrid, *et al.*, "Novel dynamic partial reconfiguration implementation of k-means clustering on FPGAs: Comparative results with GPPs and GPUs," *Int. J. Reconfig. Comput.*, 2012.