Cooperative Software-hardware Acceleration of K-means on a Tightly Coupled CPU-FPGA System

TAREK S. ABDELRAHMAN, The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Canada

We consider software-hardware acceleration of K-means clustering on the Intel Xeon+FPGA platform. We design a pipelined accelerator for K-means and combine it with CPU threads to assess performance benefits of (1) acceleration when data are only accessed from system memory and (2) cooperative CPU-FPGA acceleration. Our evaluation shows that the accelerator is up to $12.7\times/2.4\times$ faster than a single CPU thread for the assignment/update step of K-means. The cooperative use of threads and FPGA is roughly $1.9\times$ faster than CPU threads alone or the FPGA by itself. Our approach delivers $4\times\text{--}5\times$ higher throughput compared to existing offload processing approaches.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems; Reconfigurable computing**; • **Information systems** → *Clustering*; • **Computing methodologies** → Shared memory algorithms;

Additional Key Words and Phrases: Heterogeneous acceleration, FPGA-based acceleration, K-Means clustering, shared-memory CPU-FPGA systems, Performance evaluation

ACM Reference format:

Tarek S. Abdelrahman. 2020. Cooperative Software-hardware Acceleration of K-means on a Tightly Coupled CPU-FPGA System. *ACM Trans. Archit. Code Optim.* 17, 3, Article 20 (August 2020), 24 pages.
<https://doi.org/10.1145/3406114>

1 INTRODUCTION

There has been recent interest in using Field Programmable Gate Arrays (FPGAs) as compute accelerators. This interest is driven by the need for high-performance energy-efficient computing, particularly in data centers [32]. Several vendors have introduced systems that integrate FPGAs with multicore processors for this purpose [6, 24, 31, 34]. One particular class of these systems tightly couples the FPGA with the processor (or CPU) to share system memory. A circuit on the

Extension of Conference Paper: T. S. Abdelrahman, "Accelerating K-Means Clustering on a Tightly Coupled Processor-FPGA Heterogeneous System," *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016. This article extends the conference paper by (1) adding an accelerator for the update step of K-means and modifying the acceleration strategy to use it; (2) adding and validating a performance model to determine optimal load factors; (3) updating experimental evaluation on a newer generation system and additional data configurations; (4) drawing new insights from results; and (5) adding more comparison to state-of-the-art.

This work was made possible by a grant from the Intel HARP program, which provided access to a Xeon+FPGA platform. Authors' address: T. S. Abdelrahman, The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Rd, Toronto, Ontario, M5S 3G4, Canada; email: tsa@ece.utoronto.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1544-3566/2020/08-ART20

<https://doi.org/10.1145/3406114>

FPGA is able to directly read from and write to memory in a manner that is coherent with the CPU's caches. Examples of such tightly coupled systems include ones from Intel [6], IBM [31], and Xilinx [34].

This tight coupling of the FPGA and the CPU opens up new opportunities for compute acceleration with FPGAs. It is no longer necessary to employ the typical *offload* model of processing. In this model, data are copied from system memory to the FPGA device memory, processed, and then results are copied back to system memory, often leaving the CPU idle. Rather, data can remain in system memory, accessible by both the FPGA and the CPU. This enables the use of both FPGA hardware and CPU threads to concurrently and cooperatively accelerate a computation, sharing data in system memory.

Nonetheless, such tight coupling gives rise to a number of concerns. First, can an accelerator that directly accesses data from system memory deliver high performance? Second, can this accelerator be combined with CPU threads to cooperatively accelerate computation, with no explicit management of shared data? Finally, can this cooperative CPU+FPGA acceleration result in better performance of the FPGA by itself or the CPU by itself?

In this article, we address the above concerns in the context of one tightly coupled system: the Intel QuickAssist Xeon+FPGA platform and one representative application—K-means data clustering. The Intel platform connects a Xeon processor to an Arria 10 using QPI and PCIe bus channels. K-means is one of the most commonly used unsupervised clustering algorithms for data mining and/or big-data analytics [27]. It is computationally intensive and thus, not only can it benefit from FPGA acceleration, but also from combined CPU-FPGA acceleration. Our aim by using K-means is not to determine if combined CPU+FPGA acceleration is the most appropriate platform for its acceleration, but rather to use the application to explore the aforementioned concerns.

We design a pipelined accelerator for the assignment and update steps of K-means in an *Accelerator Function Unit* (AFU) on the Intel QuickAssist Xeon+FPGA platform. We then utilize this AFU in an acceleration strategy that combines CPU threads and the AFU to concurrently and cooperatively accelerate K-means. The strategy leaves all K-means shared data in system memory, allowing both the CPU threads and the FPGA equal access to these data. We develop a simple model, based on the speedup of the AFU over processor threads, to determine the best division of workloads between the processor and the FPGA.

Experimental evaluation shows that the accelerator delivers high performance, providing a speedup of up to $12.7\times$ for the assignment step and up to $2.4\times$ for the update step, compared to a single CPU thread. Further, the combined use of software threads and the FPGA hardware provides up to roughly $1.9\times$ improvement over the CPU threads by themselves or the AFU by itself, when the workloads are balanced between the CPU and the AFU. This combined use delivers up to $5\times$ higher processing throughput compared to existing offload processing approaches. These results demonstrate the utility of combined hardware and software acceleration on tightly coupled CPU+FPGA systems.

Thus, this article makes the following contributions:

- The design, implementation, and evaluation of an FPGA accelerator for K-means that accesses data directly from system memory, with no offload processing, and delivers a performance improvement of up to $12.7\times$ for the assignment step of K-means and up to $2.4\times$ for the update step.
- The design, implementation, and evaluation of an acceleration strategy that combines software threads on the CPU with the FPGA hardware to accelerate an application, sharing data in system memory. To the best of our knowledge, this is the first concurrent use of CPU threads and FPGA hardware to accelerate K-means clustering.

- An experimental demonstration of the performance improvement that cooperative use of software threads and hardware can achieve an overall performance that is almost 2X more than that of using CPU threads alone of the FPGA alone.

The remainder of this article is organized as follows: The K-means computation is described in Section 2. Our target architecture is overviewed in Section 3. The CPU-FPGA cooperative acceleration of K-means is described in Section 4. The design of the FPGA accelerator for K-means computations is presented in Section 5. Experimental evaluation is given in Section 6 followed by a review of related work in Section 7. Finally, Section 8 gives some concluding remarks and directions for future work.

2 K-MEANS CLUSTERING

K-means clustering is an unsupervised machine learning technique that is widely used for data analysis in a variety of domains [33]. It groups a set of n d -dimensional data points into K partitions, or *clusters* such that points in each cluster are similar to one another, based on some metric, but are dissimilar to points in other clusters. The technique is computationally intensive, particularly when the number of data points and/or their dimensionality is large [20, 27].

The technique groups the data points by generating an initial set of K clusters, often by randomly choosing K data points as the centroids of these clusters. It then proceeds iteratively. In each iteration, the distance between each point and each of the centroids is computed. Each point is then *assigned* to the cluster whose centroid is closest. The centroid of each cluster is *updated* based on the points that are assigned to it. This *assignment* and *update* steps are repeated until the centroids no longer change [20].

There are several ways to define the distance between a point and a centroid. A common definition is the *Euclidean* distance. Let a point be defined by the vector $\vec{p} = (p_1, p_2, \dots, p_d)$ and a centroid be defined by the vector $\vec{c} = (c_1, c_2, \dots, c_d)$. The Euclidean distance between $\vec{p}$ and $\vec{c}$ is then defined as

$$e_dist(\vec{p}, \vec{c}) = \sqrt{\sum_{i=1}^d (p_i - c_i)^2}. \quad (1)$$

Once points have been assigned to clusters, the centroid of each cluster is computed as the average of the points assigned to it. Let the set S_k be the set of points that are assigned to cluster k . The centroid of the cluster is

$$\vec{c} = \frac{1}{|S_k|} \sum_{\vec{p} \in S_k} \vec{p}. \quad (2)$$

The pseudocode of K-means is shown in Figure 1. It takes as input the number of clusters K and an array of n points. It produces as output an array ClusterID that contains the ID of the cluster each point is assigned to as well as centroids, which are the centroids of the K clusters. The code also uses two arrays sums and counts to store, respectively, the sum and number of points that belong to each cluster.

3 TARGET PLATFORM

We target the Intel QuickAssist Xeon+FPGA platform, the architecture of which is depicted in Figure 2. This platform consists of a multicore Xeon processor and an Arria 10 FPGA that both share access to memory using a number of physical channels that include Intel's Quick Path Interconnect (QPI) and two PCIe 3.0 x8 links, as shown in the figure.

The FPGA contains an Intel proprietary soft-circuit (labeled the *IP* in the figure) that provides a cache-coherent interface, called CCI-P [12], between a user's *Accelerated Function Unit* (AFU) and

```

kmeans(K, points[1:n], clusterID[1:n], centroids[1:K]) begin
  array sums[1:K], counts[1:K]
  // Initial centroids
  centroids[1:K] = points[rand()]
  more_iters = true
  while (more_iters)
    // Assignment Step
    for p = 1 to n
      min_dist = max_int
      for c = 1 to K
        dist=e_dist(points[p],centroids[c])
        if (dist < min_dist) then
          min_dist = dist
          clusterID[p] = c
        end if
      end for
    end for
    // Update Step
    sums[1:K] = 0
    counts[1:K] = 0
    for p = 1 to n
      sums[clusterID[p]] += points[p]
      ++counts[clusterID[p]]
    end for
    oldcentroids[1:K] = centroids[1:K]
    centroids[1:K] = sum[1:K]/counts[1:K]
    // Termination
    if (e_dist(centroids,oldcentroid) < thresh)
      more_iters = false
    end while
  end

```

Fig. 1. The K-means pseudocode.

the interconnect fabric. This interface allows an AFU to read from system memory by specifying the address to read from, and to write to, system memory by specifying the address to write to and the data to be written.

All memory transactions are at the granularity of a cache line and are cache-coherent with the CPU caches, as indicated by the coherence domain in the figure. The IP directs read/write requests to one of the three physical channels to best utilize bandwidth [12]. The IP further maintains a small 64 Kbytes cache that is used when accesses are made over the QPI channel. This cache is coherent with the Xeon's caches.

A number of logic layers define memory access properties [9]. For example, the *VTP* layer provides support for virtual address translation, enabling the AFU to issue read/write requests using virtual addresses. Similarly, the *RO* layer enables read response ordering so responses are received in the same order requests are issued by the AFU.

An AFU is utilized by software using Intel's Open Programmable Acceleration Engine (OPAE) [10, 21]. It defines protocols for initializing accelerators, creating a virtual address space shared between the CPU and accelerators as well as communicating with accelerators. In particular, OPAE creates and provides routines for accessing a set of *Control/Status Registers (CSRs)*. These registers are used to start an AFU, send the AFU its parameters, and otherwise communicate with and control the AFU.

Fig. 2. The architecture of the target platform (after Intel literature [12]).

A typical usage scenario has a CPU thread responsible for AFU control. This thread creates a shared virtual address space between the CPU and the FPGA, initializes the AFU, writes the (virtual) addresses of shared data to CSRs, starts the AFU, and then waits for it to be done, polling on a memory variable. Once started, the AFU reads its CSRs, thus getting pointers to the shared data. It then uses these pointers to read and write the shared data. When the AFU is done, it writes to the CSRs and/or the shared memory location the control thread is polling on. This signals the control thread to continue execution.

The ability of both the CPU cores and the AFU to read and write memory at the granularity of cache lines in a coherent manner facilitates fine-grain sharing of data between the FPGA and the CPU cores. Thus, effectively allowing both to be simultaneously used in the same computation. This opens up the opportunity of concurrent use of CPU threads and hardware for acceleration, which is what we explore in this work.

4 ACCELERATION STRATEGY

We use an AFU to accelerate both the assignment and the update steps of K-means. In each step, we partition the computations between the CPU threads and the AFU, and have both concurrently perform the K-means computations. This stands in contrast to previous work on FPGA acceleration of K-means, which either offloads the assignment step to the FPGA [5, 14] or performs both steps entirely on the FPGA [4, 7, 8, 18]. In either case, data are explicitly copied from a host's memory to the FPGA and vice versa, and the CPUs remain idle while the FPGA is active.

We utilize the shared memory between the CPUs and the AFU to store the K-means data, including the arrays that hold the input data points, the centroids, and the cluster IDs. This allows multiple CPU threads and the AFU to participate concurrently in performing the clustering computations, sharing data in the process without the need to explicitly copy data.

The overall acceleration strategy is shown in Figure 3. The main thread ($thread_0$) reads the input data, initializes the AFU and allocates the shared memory between the CPU threads and the AFU using the OPAE framework [21]. The thread then spawns t more threads. The last of these threads (labeled the *AFU thread* in the figure) is responsible for controlling and interacting with the AFU and does not participate in computations. A barrier is used to ensure that all threads start together.

The t CPU threads and the AFU thread concurrently perform the iterations of the K-means computations. In the assignment step, the data points are logically partitioned between the CPU threads and the AFU. The fraction of points assigned to the AFU is referred to as the *AFU assignment load factor*, f_a . The remaining points are equally partitioned among the t threads executing on the CPU. Thus, the AFU performs the distance calculations for $n \cdot f_a$ data points, where n is

Fig. 3. The acceleration strategy.

the total number of input points. In contrast, each CPU thread performs the same calculations on $n \cdot (1 - f_a)/t$ points. The AFU thread signals the AFU to begin execution and then waits for the AFU's done flag to be raised. Each CPU thread and the AFU produce the cluster ID of each point in the portion of points assigned to it. The threads then synchronize through a barrier before the update step is started.

In the update step, the data points are logically re-partitioned among the t CPU threads and the AFU. The fraction of points assigned to the AFU is defined by the AFU *update load factor*, f_u . The remaining points also are equally partitioned among the t threads executing on the CPU. Each thread and the AFU computes the contribution of each data point in its partition to the average of the points in a cluster. This is done in separate copies of the sums and counts arrays, one for each thread. After a barrier, $thread_0$ tallies the sums and counts arrays of all the threads, updates the centroids, and determines if more iterations are needed.

Once an iteration is over, the data are logically repartitioned again among the CPU threads and the AFU, based on the AFU assignment load factor, and the above process is repeated.

The abundant parallelism that exists in K-means lends itself to the above relatively straightforward parallelization strategy. Thus, the key challenge here is not in the strategy itself, but rather in the heterogeneous processing. In other words, the challenge is in addressing the concerns of utilizing CPU threads along with a hardware accelerator in parallelization.

The choice of the AFU assignment and update load factors should be made so that, in each step, the CPU threads and the AFU take equal amounts of time, thus balancing the workloads and minimizing idle time. These “optimal” load factors are functions of the compute power of the AFU relative to the CPU threads for each step. They can be determined using the following simple model for performance, which relies only on the speedup of the AFU over t processor threads.

Denote the speedup of the assign AFU with respect to t CPU threads by $s_a(t)$ and the un-accelerated assignment step execution time by $T_a(t)$. Since the amount of computation per data point is the same, the accelerated execution time of the assignment step can be expressed as:

$$(1 - f_a) * T_a(t) + f_a * T_a(t)/s_a(t).$$

The assignment time is minimized when the two terms in the above equation are equal. Thus,

$$(1 - f_a) * T_a(t) = f_a * T_a(t)/s_a(t).$$

Solving for f_a yields

$$f_a = s_a(t)/(1 + s_a(t)). \quad (3)$$

A similar model for the update step results in the optimal update load factor, f_u as:

$$f_u = s_u(t)/(1 + s_u(t)), \quad (4)$$

where $s_u(t)$ is the speedup of the update AFU with respect to t CPU threads.

4.1 Data Sharing

The sharing of data between the CPU and the AFU gives rise to traffic caused by data movement and coherence between the two devices. In the assignment step of clustering, $thread_0$ initializes/updates the centroids array, which is then read by other CPU threads and the AFU. However, the size of this array is typically small, leading to negligible effects on performance.

The points array is only read by the CPU threads and the AFU, in both the assignment and the update steps; thus, accesses to it generate no coherence traffic (except in the first iteration). Further, reads to this array are performed sequentially, leading to high spatial data locality. Since the input data are typically larger than the capacity of the AFU's cache, particularly when the load factor is high, it is unlikely that the points will remain in the AFU's cache across iterations. In contrast, the points array will likely remain in the larger L3 cache of the processor. However, as the input data points are logically re-partitioned among the CPU threads and the AFU in the transitions from the assignment and the update steps, L1 and L2 cache misses are likely incurred.

In the assignment step, the AFU must write its portion of `clusterID` array, requiring it to be invalidated in the processor's caches. However, given the computationally intensive nature of the assignment step, it is unlikely that there is a cost to data sharing during this step.

In the update step, the `clusterID` array and the arrays that hold the partial sums and counts are shared between the CPU threads and the AFU. Once the AFU is done computing its portion of the `clusterID` array and the data are logically re-partitioned among the CPU threads, and some of these threads must read elements of `clusterID` that have been written by the AFU, either from the AFU's cache or from memory. This requires both coherence/data traffic, since these data are stale in the processor's caches. Thus, data sharing is more likely to impact this step of K-means.

Our approach of leaving K-means data in shared memory gives rise to the above data sharing. Thus, we seek to show that performance improvements can be achieved in spite of the resulting traffic, alleviating the need to explicitly manage/partition shared data.

5 THE ACCELERATED FUNCTION UNIT

The overall organization of the AFU is shown in Figure 4. It consists of a read engine, a set of processing pipelines, and a write engine. The read engine interfaces with CCI-P and issues read requests to data initialized by the CPU. When the responses to these requests are received, they are fed to one or more of the processing pipelines. The results computed by the pipelines are fed to the write engine, which stores these results in system memory. The AFU is initialized by the software using a set of control/status registers that allow the software to pass parameters to the AFU, including the number of input data points, the start and end indices of the points assigned to the AFU, and the addresses of the various data array (e.g., points, centroids, and `clusterID`; refer to Figure 1).

The processing of the data and the results produced by the pipelines depend on the step of K-means: assignment or update. The design and operation of the assignment and update pipelines are discussed in turn.

Fig. 4. The K-means AFU architecture.

5.1 The Assignment Pipeline

During the assignment step, the read engine begins by issuing read requests to the centroids array, initialized by the CPU. Responses to these requests are stored in the centroids buffer in the AFU. The read engine then proceeds to issue read requests for the elements of the points array assigned to the AFU. Responses to data point reads are collected in a data points FIFO. The requests are pipelined. A request is issued every cycle until all the points are read, as long as the CCI-P's read request queues and the data points FIFO are not full.

On every cycle, a set of data selectors inject points from the data points FIFO into the distance pipelines. Each pipeline computes the distances between a point and all the K centroids in parallel. This is done using an array of 32-bit single-precision floating point subtractors, multipliers, and adders, as shown in Figure 5 for an example of four-dimensional data and two centroids. The subtractors compute the distance between the point and the centroid for each dimension of the data. The multipliers square this difference. The adder tree sums the squares of the distances. The units are fully pipelined and can accept input every cycle.

The resulting K distances for each point are fed to a *min* pipeline that compares the distances and determines the distance and the ID of the cluster that is nearest to the data point. This is done using a comparator tree, an example of which is shown in Figure 6 for four centroids. Each comparator takes as input a pair of distances produced by the distance calculations in addition to the index of the corresponding centroid. It produces the smaller of the two distances as well as the index of the smaller distance. Similar to the distance computations pipeline, the comparators are fully pipelined and an ID can be produced on every clock cycle.

The IDs produced by the min pipeline are stored in a cluster IDs FIFO. The write engine reads cache lines from this buffer and interfaces with CCI-P to write the IDs to the ClusterID array in

Fig. 5. Distance computation for four-dimensional data and two centroids (c_j^i denotes the j th dimension of centroid i).

Fig. 6. Minimum distance tree for two-dimensional data and two centroids.

memory, one cache line at a time. The writes are also pipelined and a write request is issued as long as the CCI-P's write queues are not full and the cluster IDs FIFO is not empty.

The operation of the read engine, the processing pipelines, and the write engine are synchronized through the FIFOs and using back-pressure signals. The write engine stalls the processing pipelines if the cluster IDs FIFO or the CCI-P's write queues become full. Similarly, the processing pipelines stall the read engine when the data points FIFO becomes full.

Once all the data points have been processed and all the cluster IDs have been written, the AFU signals the CPU that it has completed an assignment step. It then waits for the CPU to begin the update step of K-means. If a new iteration is needed, the AFU begins another iteration by reading the updated centroids again.

The performance improvement achieved by the assignment pipelines stems from performing the distance calculations in parallel. Each pipeline performs the distance calculations for a single point for all the centroids in parallel. Multiple pipelines perform these distance calculations for multiple points in parallel. Thus, the performance improvement of the assignment pipelines is proportional to:

$$p \times d \times K, \quad (5)$$

where p is the number of pipelines, d is the input data dimensionality, and K is the number of centroids. Clearly, this improvement increases with the number of pipelines, the number of

Fig. 7. The update pipeline.

dimensions, and the number of centroids. Of course, this improvement is also a function of both the CCI-P interface and memory performance. Nonetheless, it remains proportional to $p \times d \times K$.

5.2 The Update Pipelines

During the update step, the read engine reads points and their corresponding cluster IDs that were determined during the assignment step. Memory is read in cache lines, and therefore, there are several points/cluster IDs on a line. Each point and its corresponding cluster ID is dispatched to an update pipeline.

Each update pipeline maintains K partial sums, one for each of the centroids. Since the dimensionality of a centroid is d , each partial sum is a d -vector of values. The pipeline adds the point to the partial sum indicated by the cluster ID of the point. This is done using a decoder that is driven by the cluster ID. The outputs of the decoder are used as the enable signals to an array of $d \times K$ accumulators (a set of d accumulators for each of the K centroids), as shown in Figure 7, for the example of two-dimensional data and four centroids. Each accumulator corresponds to one of the centroids for each dimension of the data point. Thus, depending on the value of the cluster ID, the data point is accumulated to only one cluster's running partial sum.

Once all the data points and their cluster IDs have been processed, the K partial sums in each pipeline reflect the contribution the points fed to the pipeline make to the sum in Equation (2). Since there are multiple pipelines, the partial sums of each pipeline must be accumulated before the new centroids are determined. This accumulation can be implemented in hardware, at the cost of additional resources. However, given the small amount of time this accumulation takes, we opt for the pipelines to write their partial sums to system memory (using the write engine) and leave it to software to accumulate the partial sums of the multiple pipelines before updating the centroids. We believe this approach represents a favorable tradeoff between hardware complexity and software.

The AFU signals the CPU that it completed an update step and thus an iteration of K-means. The CPU determines if more iterations are needed and either signals the AFU to start a new assignment step or shuts down the AFU.

The performance improvement of the update pipelines essentially stems from the accumulation of points to their corresponding clusters. Thus, the performance improvement of the AFU is proportional to the number of update pipelines used. Thus, it is expected that the speedup of the update pipelines be less than that of the assignment pipelines, justifying the need for different

load factors for the assignment and updates steps. Similar to assignment, the speedup of the update pipelines is also a function of the CCI-P interface and memory performance, but it remains proportional to the number of pipelines.

6 EVALUATION

6.1 Platform

We utilize a Heterogeneous Research Architecture Platform (HARP) machine made available by Intel for academic research use.¹ The specific machine we use utilizes a 14-core Xeon E5-2680 CPU clocked at 2.4 GHz. Each CPU core has a 32 KB L1 cache and a 256 KB L2 cache. The cores share a 35 MB L3 cache. The machine also utilizes an Intel Arria 10 GX1150 FPGA (10AS016C3U19E2LG). This FPGA contains over 1.1M Adaptive Logic Modules (ALMs), over 6 MB of on-chip memory as well as over 1,500 DSP blocks. The machine has 64 GB DDR3 main memory.

The HARP platform provides an Intel proprietary soft-circuit, referred to as the *blue bitstream*. It interfaces a user's AFU (referred to as the *green bitstream*) to the interconnect fabric. This interface includes a Core Cache Interface (CCI) abstraction that exposes three physical channels to system memory (one QPI and two PCIe) as a single, multiplexed interface. There are also a number of Memory Property Factory (MPF) layers that define additional functionalities for accessing memory, such as virtual-to-physical address translation and ordered responses [9]. Finally, it implements a 64 KByte cache on the FPGA, with a cache line size of 64 Bytes, which matches that of the CPU caches.

The CPU cores read main memory with a peak bandwidth of 25.6 GB/sec, shared among all the cores. In contrast, the FPGA accesses main memory with a theoretical peak read bandwidth of 15.6 GB/sec. The latency of memory reads by the FPGA depends on the location of data. Hits to the FPGA cache have a latency of about 60 nanoseconds. Misses to data in main memory have a latency of about 355 nanoseconds, while misses to data in the CPU cache have a latency of about half this time [3].

The K-means AFU is implemented in SystemVerilog and can be instantiated for different numbers of input data dimensions, numbers of centroids, and numbers of assignment and reduce pipelines. The circuit is combined with the blue bitstream and is synthesized using Quartus Prime 64-bit version 16.0.0 Pro Edition. The circuit is clocked at 200 MHz.

An optimized software version of K-means is implemented in C++. This version is extended to use the AFU and it is used as a baseline for computing the speedup of AFU-assisted execution. This software implementation is multi-threaded, supporting the use of multiple CPU threads. We explicitly bind the CPU threads to processor cores. Thus, the maximum number of threads used is the number of cores, 14. The C++ program is compiled with g++ v 4.8.5 with optimization level -O3.

6.2 Data

Data from the UCI Machine Learning Repository [17] are used to evaluate the performance of K-means. Specifically, the household electric power consumption measurements are used. The data contain over 2M individual measurements.² Each measurement has nine attributes, which constitute the dimensions of the data. Of these nine dimensions, we use the `global_active_power` and `global_reactive_power` fields for two-dimensional data and, in

¹All results in this publication are generated using pre-production hardware and software and may not reflect the performance of production or future systems.

²Some entries are invalid, leaving approximately 2M usable ones.

Table 1. Resource Consumption and Utilization

	Data Dimensionality $\times$ Number of Clusters ($d \times K$)			
	2×8	2×16	4×8	4×16
Assignment Pipelines	8	8	4	4
Update Pipelines	16	16	16	8
ALMs	95,338 (22%)	109,681 (26%)	102,711 (24%)	104,866 (25%)
Registers	115,623 (7%)	131,168 (8%)	125,109 (7%)	127,199 (7%)
Block RAM bits	3,249,680 (6%)	3,249,680 (6%)	3,249,680 (6%)	3,249,680 (6%)
RAM Blocks	403 (15%)	403 (15%)	403 (15%)	403 (15%)
DSP Blocks	576 (38%)	1,152 (76%)	864 (57%)	1,216 (80%)

addition, use the `Sub_metering_1` and `Sub_metering_2` fields for four-dimensional data. We cluster these data into 8 and 16 clusters.

6.3 Metrics

We measure performance using several metrics. The *clustering time* is the overall time to perform the clustering, and it includes all steps of the clustering excluding the time to read the input data, spawn threads, initialize the AFU (if used), and write the output. The *clustering speedup* for t CPU threads is defined as the ratio of the clustering time of the software-only execution to the clustering time of the AFU-assisted execution, also with t CPU threads. The *assignment time* and *assignment speedup* are defined analogously, but for only the assignment step of K-means. Similarly defined are the *update time* and *update speedup* for the update step of K-means. For each experiment, we execute the code seven times, discard the lowest and highest measurements, and average the rest.

6.4 Results

All experiments are conducted on the target system described in Section 6.1. We vary the number of CPU threads between 1 and 14. In combination with the CPU threads, we utilize or do not utilize the AFU to accelerate the assignment or the update steps.

6.4.1 Resource Usage. The resource usage of the AFU circuit, which includes the blue bitstream, is shown in Table 1, for the following combinations of data dimensionality (d) and number of clusters (K), denoted by $d \times K$: 2×8 , 2×16 , 4×8 , and 4×16 . The percentages in parentheses indicate the utilization of a resource. The table also shows the number of assignment and update pipelines used for each $d \times K$ combination. The table reflects that the AFUs use a small portion of ALMs, registers, and memory on the chip. There is more significant use of the DSP blocks, which are used for the 32-bit single precision floating-point adders, subtractors, and multipliers. Thus, the number of DSP blocks on the chip is the limiting resource.

6.4.2 AFU Speedups. Figure 8 shows the speedups of the AFU over a single CPU thread, for the various $d \times K$ combinations, using 2M data points. The speedups are shown for the assignment step, the update step, as well as the overall clustering. The figure shows that the AFU provides considerable speedup for the assignment step, up to almost 12.7 $\times$. For the update step, the AFU is up to 2.4 $\times$ faster than a single CPU thread. For the overall clustering, the AFU provides a speedup that is up to 9.5 $\times$. This demonstrates the effectiveness of our AFU design in accelerating the K-means computations. There are a number of observations that can be drawn from the figure.

The figure clearly shows that the speedups achieved are a function of the dimensionality of the data, the number of clusters, and the number of pipelines. For example, the assignment step

Fig. 8. AFU assignment, update, and overall speedups over a single CPU thread for combinations of $d \times K$.

speedup for the 2×16 configuration is almost twice as large as the speedup for the 2×8 one. This is due to the larger number of parallel floating point operations performed by the AFU for the former configuration compared to the latter.

The number of parallel operations in the assign step is proportional to the number of pipelines, the data dimensionality, and the number of centroids, as indicated by Equation (5). Thus, for the same number of pipelines, and the same data dimensionality, the speedup roughly doubles when the number of centroids is doubled. This is reflected in Figure 8 for both the 2×16 and the 4×16 configurations.

The same can be observed for the speedup of the update step. In this case, the number of parallel operations is independent of the data dimensionality and the number of centroids but is proportional to the number of pipelines. Thus, the speedup for the 4×16 case is less, because 8 reduce pipelines are used instead of 16 (due to DSP constraints).

Figure 9 shows the speedup of the AFU over multiple CPU threads for combinations of $d \times K$, again using 2M data points. The goal is to observe how many CPU threads are actually equivalent in execution to the AFU, since the speedup from using t threads is not linear (i.e., less than t). The figures reflect that for the assignment step, the AFU is computationally equivalent to roughly 8 CPU threads for the 2×8 and 4×8 combinations. For the 2×16 and 4×16 combinations, the AFU is 17% and 33% faster than 14 CPU threads, the maximum that we use and which correspond to the number of cores in the CPU. In contrast, for the update step, the AFU is roughly equivalent to 4–5 CPU threads for all configurations except for the 4×16 , where it is equivalent to 2 threads.

6.4.3 Data Size. Figure 10 shows the speedup of the AFU over a single CPU thread for various input data sizes ranging from 200K points to 8M points.³ The figure shows that for sufficiently large data sizes, the speedup of the AFU is only slightly affected by the number of points being clustered. However, this is not the case when the number of points is small, where the speedup is higher. This variation is explained by the availability of data in the CPU and the FPGA caches when the data size is small.

During initialization, the CPU loads the input data, making them present in the CPU's cache. Since the input data are read-only, they remain in the cache of the CPU when the input size is small, even across the iterations of clustering. The AFU also reads a portion of these data, fetching it from the CPU's caches, which is faster than fetching it from system memory. When the input data size is large, less of the input data remain in the CPU's caches and the FPGA is more likely to read

³Data larger than 2M points were generated by duplicating the existing 2M points.

Fig. 9. AFU assignment, update, and overall speedups over CPU threads.

its portion of the data points from memory, which is slower. Further, the FPGA cache, while small, holds more of the data when the size of the input is small. This also contributes to the observed higher performance improvement at small data sizes.

To validate the above hypothesis, we flush the CPU's cache after loading the input data in memory. We confirm that the performance improvement seen in Figure 10 for small data sizes disappears with this flushing. In the remainder of this section, we present all our results using 2M points.

6.4.4 CPU+FPGA Speedups. In this section, we examine the speedups achieved when both CPU threads and the AFU are cooperatively used for processing. We first examine the impact of the assignment load factor, then the impact of the update load factor, and finally determine the speedup that is obtained for the overall clustering at the optimal load factors.

Figure 11 shows the clustering time for combinations of $d \times K$ for different numbers of CPU threads, as a function of the AFU assignment load factor, f_a . We use 1, 2, 4, and 8 threads for the 2×8 and 4×8 combinations of $d \times K$, while we use 8, 10, 12, and 14 threads for the 2×16 and 4×16 combinations. This is because the speedup of the AFU for the assignment step for the former combinations is smaller than that for the latter. Thus, to expose the impact of the load factor, we use a higher number of threads for the latter combinations. In the figure, the update load factor is set to 1.0.⁴

⁴We experimentally validated that the update load factor has little impact on the execution time results for the assignment step. Thus, the update time in this experiment is a constant and only the assignment step impact clustering time.

Fig. 10. AFU assignment, update, and overall speedups over a single CPU thread for various input data sizes.

The following observations can be made from Figure 11, and they equally hold for all the combinations of $d \times K$. The clustering time initially declines as the assignment load factor is increased. This clearly is due to CPU threads offloading increasing amounts of assignment computations to the AFU. This decline continues until the time taken by the AFU to perform the assignment step is roughly equal to that of the CPU threads. Further increasing f_a causes the AFU to take longer than CPU threads, leaving them idle and leading to an increase in the combined assignment time. The *optimal* values of f_a that achieve the balance in computation time between the CPU threads and the AFU depend on the number of CPU threads used and the $d \times K$ configuration. For example, for 2×8 , the optimal values of f_a are roughly 0.9, 0.8, 0.7, and 0.6 for 1, 2, 4, and 8 threads, respectively.

The above optimal load factors reflect the ones predicted by Equation (3) in Section 4. Using the speedups of the AFU for the 2×8 combination gives optimal f_a values of 0.87, 0.83, 0.76, and 0.57. These values correlate closely with those observed experimentally. The same holds for the other combinations of $d \times K$. This indicates that once the speedup of the AFU is determined, it is possible to predict the optimal load factor for the assignment step.

Figure 12 shows the clustering time for combinations of $d \times K$ for different numbers of CPU threads, as a function of the AFU update load factor, f_u . Unlike the assignment step, the speedup of the AFU for the update step is $2\times$ – $2.5\times$ and thus, we limit the number of threads to 1, 2, 4, and 8. These numbers of threads are sufficient to expose the impact of the update load factor. In the figure, the assignment load factor is set to 1.0, and similar to above, we experimentally validated that the assignment load factor has little impact on the execution times of the update step. Thus, in this experiment, the assignment time is constant and only the update steps impact clustering time.

Fig. 11. Clustering time as a function of the assignment load factor for different threads.

Similar observations to those made for the assignment step can be made here. The update time initially declines as the update load factor, f_u , is increased and the CPU threads offload more of the update computations to the AFU. This continues until the amount of update computations performed by the CPU threads and the AFU are equal. The update time starts to increase with more computations assigned to the AFU, which leaves the CPU threads idle. In the case of the 2×16 combination, the optimal update load factors are observed as roughly 0.75, 0.6, 0.5, and 0.4 for 1, 2, 4, and 8 threads, respectively. Once again, these observed values correlate well with those predicted by Equation (4) in Section 4, namely, 0.70, 0.64, 0.52, and 0.38.

Finally, we examine the speedup of the entire clustering. The above results suggest that different assignment and update load factors must be used. The optimal load factors for each step can be determined either from the Equations (3) and (4) or from the experimental evaluation above. Nonetheless, we explore different combinations of load factors around these optimal values to determine the best combination for the overall clustering. More specifically, we sweep through combinations of load factors centered around the optimal load factors determined above.

The results of this sweep of load factors is shown in Figure 13. It depicts the overall clustering time as a function of the assignment and update load factors swept through. The horizontal axis is the combination of assignment and update load factors used, i.e., (f_a, f_u) . The same number of threads is used for both the assignment step and the update step: 8 threads for 2×8 and 4×8 , and 14 threads for 2×16 and 4×16 . This is again driven by the speedup of the AFU for the assignment step. The figure indicates that while there are variations in the clustering time for the different

Fig. 12. Clustering time as a function of the update load factor for different threads.

combinations of load factors, these variations are small. This reflects that there is little sensitivity to the load factors around the optimal values. Further, the best combination is closely reflected by the optimal load factors determined by Equations (3) and (4). Therefore, it is possible to determine how to balance the workload between the CPU and the FPGA based on the speedups of the AFUs in relation to CPU threads.

Table 2 shows the best performing combination of load factors and the speedup of the clustering at these load factors over the CPU threads by themselves and over the AFU by itself. The table shows that the clustering computations at these load factors are up to $1.8\times$ – $1.9\times$ faster than the CPU thread by itself or the AFU alone. This improvement is achieved when the workload of the CPU is almost the same as the workload of the FPGA. Thus, the best speedup that can be achieved is $2\times$, and our concurrent acceleration achieves almost this best. This demonstrates the benefits of using both the CPU and the AFU to concurrently perform the clustering, sharing a single address space across the two steps of the computation.

6.4.5 Bandwidth. The AFU shares the interconnect (QPI and PCIe) bandwidth with the processor. Thus, an important consideration when using a cooperative acceleration strategy like ours is to what extent does this concurrent use of the CPU threads and the AFU affect the performance of each. To explore this issue, we modify our software to always use the AFU (i.e., load factors of 1.0) while also using CPU threads to do the same (i.e., load factors of 0); thus, in effect, both the AFU and the CPU threads and reading and writing the same data. Since both compute the same values,

Fig. 13. Overall clustering speedup sweep around optimal load factors.

Table 2. Speedup of CPU+AFU over CPU by Itself and AFU by Itself

	2×8	2×16	4×8	4×16
Optimal (f_a, f_u)	(0.6,0.3)	(0.5,0.3)	(0.7,0.2)	(0.6,0.2)
Speedup over CPU	1.96	2.06	1.87	1.77
Speedup over AFU	1.77	1.66	1.94	1.91

we leave their writes unsynchronized as a benign data race. We then compare the execution times of the AFU and the CPU threads to those of the case in which only the CPU threads are used or only the AFU is used.

Figure 14 shows the ratio of the execution time of each of the AFU and the CPU threads when both are used as described above to their counterparts when only one is used, as a function of the number of threads used. This is only shown for the 2×8 configuration and only for the assignment step; the update step and the rest of the configurations exhibit similar results. The figure reflects that there is almost no change to the AFU execution time, while there is minimal change to the CPU execution time; well within measurement error. This indicates that while there is data sharing, as described in Section 4.1, such sharing has little impact on the overall performance of either the CPU or the AFU. This validates our acceleration strategy that leaves data in shared memory accessible by both the CPU and the FPGA.

Fig. 14. AFU assignment execution time ratios.

Table 3. Power Efficiency in MFLOPS/Watt

	MFLOPS/Watt			
	AFU Alone	CPU 8 threads	CPU 14 threads	CPU+AFU
2×8	692	157	154	312
2×16	1,261	254	279	425
4×8	703	253	287	312
4×16	1,257	368	397	436

6.4.6 Power Efficiency. A key advantage for using FPGAs to accelerate computation (in addition to reducing execution time) is improving power efficiency. Thus, we report on the power efficiency of using the AFU, both by itself and in cooperation with CPU threads.

Our target platform restricts the ability to experimentally measure energy consumption and thus, we estimate power. We use the Quartus power analysis tools [11] to obtain power consumption for the AFU. We obtain CPU power using PAPI [28]. This is done for the entire K-means computation, reflecting both the assignment and update phases. It is also done at various load factors: 0 for CPU-only, 1 for AFU-only, and the optimal factors/number of threads that lead to a balanced workload between the CPU threads and the AFU, as indicated by Figure 13. In the last case, the separate CPU and AFU energy consumptions are added to determine the overall consumption. To normalize for the different amount of computation done by each input data configuration (i.e., data dimensionality and number of clusters), we express power efficiency using the number of Mega floating point operations per Watt (MFLOPS/Watt).

K-means computations, as outlined in Section 2, perform roughly $I * (3 * K * n * d + n * d)$ floating point additions, subtractions, and multiplication, where I is the number of iterations, K is the number of centroids, n is the number of points, and d is the dimensionality of each point. With the knowledge of reported execution time, it is possible to calculate the achieved MFLOPS and thus, the MFLOPS/Watt.

Table 3 shows the MFLOPS/Watt for each AFU configuration for the CPU using 8 and 14 threads, as well as for using CPU threads and the FPGA. Two conclusions can be drawn from the table. The first is that the AFU by itself is 2.8× to 4.5× more power efficient than the CPU by itself. This demonstrates the power efficiency advantage of the FPGA. The second conclusion is that when

Table 4. Throughput (in GFLOPS) Comparisons to Existing Work

Work	n	K	d	I	GFLOPS
This article	2M	8 and 16	2 and 4	18–50	26.2–55.9
Souza et al. [26]	4,096–65,536	256–1,024	16	13–48	7.6–10.6
Intel [13]	1,024	10	2	25 [†]	14.1
Choi and So [4]	1M	6	2	50 [†]	2.5

[†]The number of iterations is not reported. We use I for our dataset with the values of n , K , and d reported.

the CPU is combined with the AFU to minimize execution time (as indicated in Figure 13), the resulting power efficiency drops in comparison to the AFU by itself but is still higher than the CPU by itself. This is expected, since the addition of CPU threads to the AFU reduces execution time by a factor of only about $1.8\times$ (see Table 2), but brings in the CPU with a $2.8\times$ – $4.5\times$ less power efficiency. Similarly, the addition of the AFU to the CPU threads reduces execution by the same $1.8\times$ factor with the more power-efficient AFU, thus reducing energy consumption compared to the CPU alone.

An interesting observation is that it may be possible to trade off execution time for energy efficiency by using higher load factors than those indicated in Figure 13. This utilizes more of the energy-efficient AFU and less of the CPU. While this increases overall execution time, it may improve combined CPU-FPGA power efficiency. The exploration of this tradeoff is left to future work.

6.5 Comparison to State-of-the-Art

Comparisons to existing work on FPGA acceleration of K-means is complicated by a number of factors. Different works use different FPGA platforms with varying compute capabilities and different FPGA-to-memory bandwidths. Further, evaluation is often presented with different datasets, with different data sizes, numbers of clusters and dimensionality, and in some cases the datasets are not fully specified. Sometimes, only speedups against a CPU implementation are reported. Existing work also uses offload processing, and in many instances, it is unclear if the time to copy data from system memory to FPGA memory is even included.

Nonetheless, we make a best-effort attempt to compare to existing work, which mostly uses offload processing. To provide a reasonably fair comparison, we use *throughput*, as measured by the number of Floating Point Operations Per Second (FLOPS). The number of floating point operations for K-means can be calculated as in Section 6.4.6. This combined with execution time enables us to calculate the FLOPS.

Table 4 compares the throughput we achieve to that achieved by recent work (see Section 7 for more details on this work), where the evaluation is detailed enough to draw comparisons. We achieve 26.2/48.3/30.5/55.9 GFLOPS for the $2 \times 8/2 \times 16/4 \times 8/2 \times 16$ combinations of d and K , respectively. These throughputs are for the combined use of the CPU and the AFU at the optimal load factors indicated by Figure 13.

We compare to the work by Souza et al. [26], which uses the same platform we use but restricts the acceleration to the FPGA (i.e., offload processing with no cooperative CPU-FPGA use). They achieve 7.6/10.1/10.3/10.5/10.6 GFLOPS for their Tiny/Small/Standard/Large/Huge datasets. Thus, we achieve up to $5.3\times$ better throughput. We also compare to work reported by Intel, which uses a similar Arria 10 FPGA to the one we use [13]. However, they copy K-means data to on-chip memory and are thus limited to small datasets. Although they report a $10\times$ speedup over a CPU implementation (comparable to the speedup we get), we achieve about $4\times$ higher throughput,

depending on the dataset, and we are not limited in the size of the datasets as they are. We finally compare to the work of Choi and So [4]. We achieve 10× more throughput for our 2×8 case, but it is important to point out that they evaluate a map-reduce implementation on a cluster and data transfers limit their performance. Thus, in summary, our approach achieves higher throughput compared to existing and similar approaches.

7 RELATED WORK

The computationally intensive nature of K-means has prompted several researchers to explore its acceleration on FPGAs. In contrast to these earlier explorations, our work is unique in that it examines the acceleration of K-means on a tightly coupled shared-memory heterogeneous system. This allows the partitioning of the workload between CPU cores and the AFU and actively uses both to perform the clustering. Although these explorations use different FPGA technologies, we perform the best-effort performance comparisons reported in Section 6.5.

Hussain et al. [7, 8] implement K-means completely in hardware and report significant speedups over a software implementation. However, their implementation uses fixed-point arithmetic and demands all data be resident in the FPGA's memory blocks. This limits them to smaller datasets of about 3K points and requires that data be explicitly moved to the memory blocks.

Lavenier [14] implements the assignment step of K-means on a systolic array in an FPGA, leaving the update to software. Performance is limited by the cost of data communication between the two phases of the computation [5]. In contrast, the same data sharing does not significantly affect the performance of our implementation.

Lin et al. [18] implement K-means for high dimensionality data using the triangle inequality optimization. They avoid frequent data transfer by performing all the cluster operations on the FPGA and using DDR3 memory to store data, but also for limited size datasets.

Choi and So [4] accelerate K-means as a case study for accelerating map-reduce computations on a heterogeneous cluster of nodes equipped with FPGAs. The cluster contains an additional dedicated communication network for the FPGAs. In contrast, we focus on tightly coupled CPU-FPGA systems. They process large datasets and report speedups of up to 20× over a Java implementation on a three-node cluster. In contrast, we achieve speedups of about 10× on a single FPGA and over a highly optimized C++ implementation.

Liu et al. [19] accelerate the update step of K-means with on-the-fly updates to the centroids, implementing a continuous update variant of the computation. They use a systolic array on an FPGA for higher performance.

Lee [15] and Lee et al. [16] propose a multilevel data clustering method that extends K-means clustering and that allows a streaming-based scalable hardware acceleration. Their methods support large number of data dimensions and number of clusters. They utilize a CPU-FPGA system that uses high-speed channels for CPU-FPGA data communication and utilize the CPU only for data shuffling before sending to the FPGA for processing. Thus, the bulk of processing is done on the FPGA with no cooperative acceleration with processor threads, as we do. They achieve a throughput of up to about 45 MSamples/s, which is lower than what we achieve. However, their use of 16-bit integers to represent data makes such direct comparison of throughputs non-indicative.

Souza et al. [26] implement K-means in OpenCL and translate it to FPGA hardware using the Intel OpenCL compiler. They target the same platform as we do, but they do not utilize the CPU threads concurrently with the FPGA; i.e., the CPU is idle while the FPGA is performing computations. Thus, they do not utilize the tight CPU-FPGA integration offered by the platform. They report up to 3× speedup for the AFU, which is less than the speedup of our AFU. They measure energy efficiency and show similar advantage for the FPGA as we do.

Intel describes a K-means FPGA accelerator that is expressed in OpenCL and compiled into an Arria 10 FPGA [13]. They store the points data in on-chip memory and thus, the size of the datasets they process is limited (up to 1,024). They also do not explore cooperative CPU-FPGA acceleration. The throughput of their approach is compared to that of ours in Section 6.5.

Kamali [22] and Kamali et al. [23] describe MUCH-SWIFT, an FPGA-based architecture for performing K-means clustering using kd-trees. They use a Zynq-7000 Ultrascale+ SoC with four ARM processor and programmable logic. A key aspect of their implementation is the use of filtering to prune the kd-tree. They employ the ARM processors to do this filtering and utilize the programmable logic to perform calculations for a point. They report up to 330 $\times$ speedup over unoptimized implementations; i.e., ones that do not use their filtering. In contrast, we utilize CPU cores in a system and use the FPGA as an additional hardware thread that equally participates in the K-means calculations.

This work extends the earlier study on the acceleration of K-means on a CPU-FPGA system [1] by adding an accelerator for the update step of K-means and modifying the acceleration strategy to use it. It further extends the evaluation to a newer generation system and explores additional input data configurations. Finally, it proposes a model for determining optimal load factors for FPGA processing and experimentally validates the model.

There has also been work on exploring shared memory CPU-FPGA systems. Weisz et al. [30] examine pointer chasing using the FPGA fabric and show that it is efficient, especially with processor assistance. Zhang et al. [35] develop high-throughput sorting on an earlier generation of the same platform we use in this work. Similarly, Zhou and Prasanna [36] explore graph analytics on the same Intel Xeon+FPGA platform we use.

8 CONCLUDING REMARKS

We present the design, implementation, and evaluation of software-hardware cooperative acceleration on systems that tightly couple a CPU and an FPGA. We target the Intel QuickAssist Xeon+FPGA platform and use K-means data clustering as a case study to assess the feasibility and benefit of building an accelerator that accesses data only from system memory with no offload processing and of cooperative acceleration.

Our evaluation shows that the FPGA accelerator delivers a performance improvement over a single CPU thread of up to 12.7 $\times$ for the assignment step, up to 2.4 $\times$ for the update step, and up to 9.5 $\times$ for the overall clustering. Further, additional performance improvements can be achieved by combining the CPU threads and the FPGA to cooperatively accelerate both the assignment and the update steps of K-means. Specifically, a speedup of about 1.9 $\times$ is achieved over using only CPU threads or only the FPGA accelerator. This performance represents an improvement in processing throughput (in floating point operations per second) that is 4 $\times$ -5 $\times$ compared to existing offload processing work. Thus, our study and its results demonstrate the effectiveness of cooperative CPU-FPGA acceleration.

There are a number of ways in which this work can be extended. First, it is possible to approximate the floating point arithmetic carried out by DSP blocks by fixed-point arithmetic, freeing DSP resources and allowing an increase in the number of AFU pipelines. This can lead to higher performance improvements. Second, this work provided some indication of the power efficiency of the concurrent use of CPU threads and the FPGA when execution time is minimized. An exploration of tradeoffs that may exist between execution time and power efficiency can be pursued. Another direction for future work is to explore heterogeneous dynamic load-balancing techniques (e.g., References [2, 25, 29]), which may be useful in spite of the regular and predictable nature of K-means computations. Finally, it is interesting to consider case studies of other applications that demand even finer-grain sharing of data between the processor and the AFU.

REFERENCES

- [1] T. S. Abdelrahman. 2016. Accelerating K-means clustering on a tightly coupled processor-FPGA heterogeneous system. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP'16)*. 176–181.
- [2] M. E. Belviranlı, L. N. Bhuyan, and R. Gupta. 2013. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.* 9, 4 (2013).
- [3] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2019. In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms. *ACM Trans. Reconfig. Technol. Syst.* 12, 1 (2019), 4:1–4:20.
- [4] Yuk-Ming Choi and Hayden Kwok-Hay So. 2014. Map-reduce processing of K-means algorithm with FPGA-accelerated computer cluster. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*. 9–16.
- [5] Maya Gokhale, Jan Frigo, Kevin McCabe, James Theiler, Christophe Wolinski, and Dominique Lavenier. 2003. Experience with a hybrid processor: K-means clustering. *J. Supercomput.* 26, 2 (2003), 131–148.
- [6] P. Gupta. 2015. *Xeon+FPGA Platform for the Data Center*. Retrieved from http://www.ece.cmu.edu/~calcm/carl/doku.php?id=pk_gupta_intel_xeon_fpga_platform_for_the_data_center.
- [7] Hanaa M. Hussain, Khaled Benkrid, Ahmet T. Erdogan, and Huseyin Seker. 2011. Highly parameterized K-means clustering on FPGAs: Comparative results with GPPs and GPUs. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*. 475–480.
- [8] Hanaa M. Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet T. Erdogan. 2011. FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering Microarray data. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS'11)*. 248–255.
- [9] Intel. 2020. *MPF—Memory Properties Factory*. Retrieved from https://github.com/OPAE/intel-fpga-bbb/tree/master/BBB_cci_mpf.
- [10] Intel Corp. 2019. *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*. Retrieved from <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf>.
- [11] Intel Corp. 2020. *Intel QuickAssist Technology*. Retrieved from <http://www.intel.com/content/www/us/en/embedded/technology/quickassist/overview.html>.
- [12] Intel Corp. 2020. *Power Solutions*. Retrieved from <https://www.intel.com/content/www/us/en/programmable/support/supportresources/support-centers/power-support.html>.
- [13] Intel Documentation. 2020. *AN 856: K-Mean Clustering with the Intel FPGA SDK for OpenC*. Retrieved from <https://www.intel.com/content/www/us/en/programmable/documentation/rgw1528307246592.html>.
- [14] Dominique Lavenier. 2000. FPGA implementation of the K-means clustering algorithm for hyperspectral images. *Los Alamos National Lab, LAUR #00-3079* (2000), 1–18.
- [15] Dajong Lee. 2017. *Designing Hardware Accelerated Systems for Imaging Flow Cytometry*. Ph.D. Dissertation. UC San Diego.
- [16] Dajong Lee, Alric Althoff, Dustin Richmond, and Ryan Kastner. 2017. A streaming clustering approach using a heterogeneous system for big data analysis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. 699–706.
- [17] M. Lichman. 2013. UCI Machine Learning Repository. Retrieved from <http://archive.ics.uci.edu/ml>.
- [18] Zhongduo Lin, Charles Lo, and Paul Chow. 2012. K-means implementation on FPGA for high-dimensional data using triangle inequality. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 437–442.
- [19] Wei-Chuan Liu, Jiun-Long Huang, and Ming-Syan Chen. 2005. KACU: K-means with hardware centroid-updating. In *Proceedings of the Emerging Information Technology Conference*. 3–5.
- [20] S. Lloyd. 1982. Least squares quantization in PCM. *Trans. Inf. Theor.* 28, 2 (1982), 129–137.
- [21] Enno Luebbers, Song Liu, and Michael Chu. 2020. *Simplify Software Integration for FPGA Accelerators with OPAE*. Retrieved from <https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf>.
- [22] Hadi Mardani Kamali. 2018. Using multi-core HW/SW co-design architecture for accelerating K-means clustering algorithm. *CoRR abs/1807.09250* (2018).
- [23] Hadi Mardani Kamali and Avesta Sasan. 2018. MUCH-SWIFT: A high-throughput multi-core HW/SW co-design K-means clustering architecture. In *Proceedings of the Great Lakes Symposium on VLSI*. 459–462.
- [24] Andrew Putnam et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*. 13–24.

- [25] A. Rodriguez, A. Navarro, R. Asenjo, F. Corbera, R. Gran Tejero, D. Suarez Gracia, and J. Nunez-Yanez. 2019. Parallel multiprocessing and scheduling on the heterogeneous Xeon+FPGA platform. *J. Supercomput.* (06 2019).
- [26] Matheus Souza, Lucas Maciel, Pedro Penna, and Henrique Freitas. 2018. Energy efficient parallel K-means clustering for an Intel hybrid multi-chip package. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*. 372–379.
- [27] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. 2018. *Introduction to Data Mining* (2nd ed.). Pearson.
- [28] Daniel Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2009. Collecting performance data with PAPI-C. In *Proceedings of the 3rd Parallel Tools Workshop on Tools for High Performance Computing*. 157–173.
- [29] A. Vilches, R. Asenjo, A. G. Navarro, F. Corbera, R. Gran Tejero, and M. Garzarán. 2015. Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. In *Proceedings of the International Conference on Computational Science*, Vol. 51. 140–149.
- [30] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. 2016. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 264–273.
- [31] Bruce Wile. 2014. *CAPi is Core to POWER*. Retrieved from <http://www-03.ibm.com/linux/blogs/capi/>.
- [32] R. Wilson. 2014. *Heterogeneous Computing Meets the Data Center*. Retrieved from https://www.altera.com/solutions/technology/system-design/articles/_2014/heterogeneous-computing.html.
- [33] Junjie Wu. 2012. *Advances in K-means Clustering*. Springer-Verlag Berlin.
- [34] Xilinx Inc. 2014. *Zynq-7000: All Programmable SoC*. Retrieved from <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>.
- [35] Chi Zhang, Ren Chen, and Viktor Prasanna. 2015. *High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform*. Technical Report CENG-2015-10. Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, California.
- [36] Shijie Zhou and Viktor K. Prasanna. 2017. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*. 137–144.

Received February 2020; revised March 2020; accepted June 2020