Accelerating K-Means Clustering on a Tightly-Coupled Processor-FPGA Heterogeneous System*

Tarek S. Abdelrahman
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
tsa@eecg.utoronto.ca

***Abstract*—We present a case study of the design of an FPGA accelerator for a tightly-coupled shared-memory processor-FPGA system: the Intel QuickAssist FPGA platform. We use K-means as an example computationally-intensive application and design a pipelined accelerator for calculating minimum distances between points and centroids. Our accelerator is unique in that it works in collaboration with CPU threads, accessing shared data in system memory. It achieves a speedup of 3.8X over a reference software implementation. Moreover, the combined use of CPU threads and the FPGA accelerator achieves a speedup of up to 2.9X over using only the CPU threads and of up to 1.9X over using only the accelerator, depending on the number of CPU threads. We analyze the impact of data sharing between the CPU threads and the FPGA and reason that while this sharing increases memory traffic, it has little impact on overall performance.**

Keywords. Heterogenous architectures; Shared-memory processor-FPGA systems; K-Means clustering; Performance evaluation.

I. Introduction

The last few years have witnessed an increased interest in the use of Field Programmable Gate Arrays (FPGAs) to accelerate computations. This interest is driven by the need for high-performance energy-efficient computing, particularly in data centers [1]. Thus, it is to no surprise that several vendors are introducing systems that integrate an FPGA with multicore processors.

An emerging class of FPGA-accelerated systems is one that tightly integrates the FPGA with the processor to share system memory. A soft circuit implemented in the FPGA fabric is able to directly read from and write to memory in manner that is coherent with the processor's caches. Examples of such tightly-coupled shared-memory systems include ones from Intel [2], IBM [3], Xilinx [4] and Convey [5].

This cache-coherent integration of processors and FPGAs has several advantages. First, it eliminates the need to explicitly copy data between the memory of the processor and that of the FPGA. Second, it allows for both the processor cores (the CPUs) and an FPGA circuit to concurrently perform a computation. Finally, it allows fine-grain sharing of data between the CPUs and the FPGA circuit, potentially enabling new classes of applications to benefit from FPGA acceleration. However, such tight integration can negatively impact performance when there is considerable data sharing between the processor and the FPGA, mainly because of the FPGA's lower operating frequency compared to that of the processor (5X to 10X slower) and to the FPGA's limited cache resources.

In this paper, we explore the design of an FPGA accelerator, using a case study, on a prototype of the Intel QuickAssist FPGA platform that connects a Xeon processor to a Stratix V FPGA using a QPI link. We use K-means clustering as our case study. K-means is a popular clustering approach that is computationally intensive when the data size is large and thus benefits from FPGA acceleration. Our design is unique in that it works in collaboration with CPU threads, accessing shared data in system memory. It allows the combined use of CPU threads and the accelerator and leads to better performance than either the threads by themselves or the FPGA circuit by itself.

We design a pipelined implementation of distance calculations of K-means in an *Accelerator Function Unit* (AFU) on the Intel platform. Experimental evaluation of clustering a large data set of 2 million points shows that the AFU is 3.8X faster than a software reference implementation. The combined use of the AFU and a single CPU thread improves performance over the single thread by 2.9X and over the AFU alone by 1.1X. With 4 CPU threads, the combined use improves over thread performance by 1.6X and over AFU performance by 1.9X. An analysis using performance counters on the processor reflects increased memory traffic due to the data sharing between the processor and the AFU. However, this traffic has minimal impact on performance.

This paper makes the following contributions:

- The design of an FPGA accelerator for the emerging class of tightly-coupled shared memory processor-FPGA systems, using K-means as a case study.
- Demonstrating the benefit of sharing system memory between CPUs and an FPGA accelerator resulting in performance improvements over the CPUs alone or the FPGA accelerator alone.

The remainder of this paper is organized as follows. The K-means computation is described in Section II. Our target platform, the Intel QuickAssist FPGA platform, is overviewed in Section III. The design of the FPGA accelerator and how it interacts with CPU threads is detailed in Section IV. Experimental evaluation is given in Section V followed by a review of related work in Section VI. Finally, Section VII gives some concluding remarks.

II. K-means Clustering

K-means clustering is an unsupervised machine learning technique that is widely used for data analysis in a variety of

*This work was made possible by an equipment donation from Intel and Altera.

domains [6]. 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 [7].

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 (defined below) 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 [7].

A common definition of the distance between a point and a centroid is the *Euclidean* distance. Let a point be defined by the vector $\vec{p} = (p_1, p_2, \cdots, p_d)$ and a centroid be defined by the vector $\vec{c} = (c_1, c_2, \cdots, c_d)$. 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

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

K-means 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.

III. Target Platform

The high-level architecture of the Intel QuickAssist FPGA platform consists of a multicore processor and an FPGA that share access to memory and I/O devices using Intel's Quick Path Interconnect (QPI). The FPGA contains an Intel proprietary soft-circuit (*QPI IP*) that interfaces a user's *Accelerated Processing Unit* (AFU) to the QPI fabric. This enables an AFU to issue read and write requests to shared memory. The QPI IP also provides for an on-chip cache and support for virtual address translation. Thus, in effect, both the CPU cores and the AFU can read and write memory at the granularity of cache lines in a coherent manner. This architecture facilitates fine-grain sharing of data between the FPGA and the CPU cores, effectively allowing both to be simultaneously used in the same computation.

An AFU is utilized by software using Intel's QuickAssist Technology [8], which defines a protocol for initializing, communicating, and sharing data with accelerators in Intel-based systems. The Accelerator Abstraction Layer (AAL) provides a set of runtime routines for the creation of an up to 2 GB shared virtual address space, for the allocation and use of a *Device Status Memory* (DSM) that is used to communicate with the AFU, and for control of the AFU.

A typical usage scenario has a CPU control thread create the shared virtual address space, initialize the AFU, write the (virtual) addresses of shared data to the DSM, start the AFU and then wait for it to be done. Once started the AFU reads the DSM, thus getting pointers to the shared data. The AFU may use these pointers to read and write shared data. When the AFU is done, it writes to the DSM, signaling the control thread to continue execution.

The Intel Hardware Accelerator Research Program has made a number of QuickAssist FPGA platforms available for academic research use. These platforms are *pre-production* systems that connect a 10-core Intel Xeon E5-2600 v2 2.8 GHz processor with an Altera Stratix V A7 (5SGXEA7N1F45C1) FPGA. The system has 32 GB of DDR3 1600 MHz DRAM. The QPI speed is 6.4 GT/s. The QPI IP implements a 64 KB cache with a cache line size of 64 bytes. An AFU read cache hit has a latency of about 25 nanoseconds, while a read cache miss to a processor's cache has a latency of approximately 180 nanoseconds [9].

IV. The Accelerated Function Unit

A. Acceleration Strategy

The assignment step is the computationally intensive part of K-means, taking 80–90% of the clustering time. Thus, we use the AFU to accelerate this step, leaving the update step to the processor. This has the added value of increasing data sharing between the processor and the AFU, allowing us to assess its potential impact on performance.

We partition the computations between the processor cores (CPUs) and an AFU and have both concurrently perform the K-means computations. This is in contrast to previous work on FPGA acceleration of K-means, which either offloads the assignment step to the FPGA [10], [11] or entirely performs both steps on the FPGA [12], [13], [14], [15]. In either case, data is explicitly copied from a host's memory to the FPGA and vice-versa and the CPU cores remain idle while the FPGA is active.

We utilize the shared memory between to the CPUs and the AFU to store the K-means data, including the arrays that hold the input 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 1. 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 AAL framework. 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. A barrier is used to ensure that all threads start together.

The t CPU threads and the AFU thread 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 *load factor*, f. The remaining points are equally partitioned among the t concurrent threads

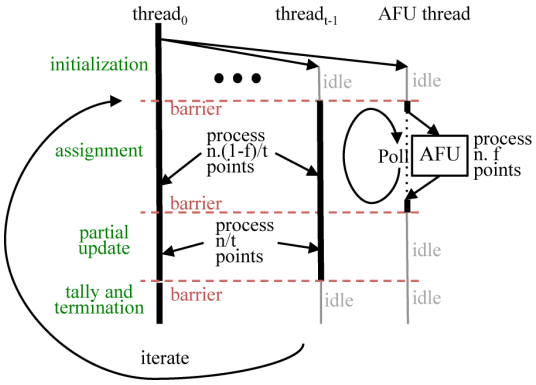

Fig. 1. The acceleration strategy.

executing on the CPUs. Thus, the AFU performs the distance calculations for $n.f$ data points, where n is the total number of input points. In contrast, each CPU thread performs the same calculations on $n.(1-f)/t$ points. The AFU thread signals the AFU to begin execution and then spins polling on an AFU's done flag. 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 AFU remains idle and the data points are logically re-partitioned among the t CPU threads, each processing n/t points. Each thread computes the contribution for each data point in its portion of the data points 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 is logically repartitioned again among the CPU threads and the AFU, based on the AFU load factor, and the above process is repeated.

The choice of the AFU load factor should be made so that the CPU threads and the AFU take equal amounts of time, thus balancing the workloads and minimizing idle time during the assignment step. This "optimal" load factor is a function of the compute power of the AFU relative to a CPU thread as well as of the number of CPU threads.

B. The AFU Design

The AFU consists of a *read engine*, a number of *distance pipelines* and *min pipelines* and a *write engine*. Upon initialization, the software writes to the Device Status Memory (DSM) of the AFU several parameters, including, the number of input data points, the start and end indices of the points assigned to the AFU, the dimensionality of the data, the number of clusters as well as the addresses of the input data array (`points`), the centroids array (`centroids`) and the cluster IDs array (`clusterID`).

The read engine interfaces with the QPI IP circuit and issues 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 buffer*. The requests are pipelined. A requests is issued every cycle until the all points are read, as long as the QPI IP's read request queues and the data points buffer are not full.

On every cycle, a set of data selectors inject points into the distance pipelines. Each pipeline computes the distances between a point and all the centroids in parallel. This is done using an array of 32-bit single precision floating point subtractors, multipliers and adders. The subtractors compute the distance between the point and the centroid in each dimension of the data. The multipliers square this difference. The adder tree sums the squares of the distances. The units are pipelined and at steady state, distances between a point and all the centroids are produced every cycle.

The resulting K distances for each point are fed to a min pipeline which 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. Similar to the distance pipeline, the floating point units and the comparators are pipelined and at steady state, an ID is produced on every clock cycle.

The IDs produced by the min pipeline are stored in a *cluster IDs* buffer. The write engine reads cache lines from this buffer and interfaces with the QPI IP to write the IDs to the `ClusterID` array in memory, one cache line at a time. The writes are also pipelined and a write request is issued as long as the QPI IP's write queues are not full and the cluster IDs buffer is not empty.

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 iteration of K-means. It then waits for the CPU to update the centroids and signal it that a new iteration is needed or that the computation is done. In the former case, the AFU begins another iteration by reading the updated centroids again.

C. Data Sharing

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. Thus accesses to it generate no coherence traffic (except in the first iteration). Further, reads to this array are performed sequentially, leading to high data locality. Since the input data is 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 25 MB L3 cache of the processor. Finally, 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 significant 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, the data is 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 this data is stale in

	4 Pipelines		8 Pipelines	
	AFU	QPI IP	AFU	QPI IP
ALMs	72,525 (31%)	67,410 (29%)	128,020 (55%)	56,725 (24%)
Registers	115,412 (12%)	49,828 (5%)	160,560 (17%)	49,825 (5%)
Block RAM bits	93,546 (0.17%)	1,030,592 (2%)	100,074 (0.2%)	1,030,592 (2%)
DSP Blocks	64 (25%)	0 (0%)	128 (50%)	0 (0%)

TABLE I
RESOURCE CONSUMPTION AND UTILIZATION.

the processor's caches. Thus, data sharing is more likely to impact this step of clustering and is the focus our evaluation of the cost of data sharing.

V. EVALUATION

A. Experimental Platform and Methodology

The K-means AFU is implemented in SystemVerilog, parameterized by the number of assignment/min pipelines, the dimensionality of the input data and the number of centroids. The circuit is combined with the Intel QPI IP and is synthesized using Quartus II 64-bit version 13.1. The circuit is clocked at 200 MHz.

The resources of the AFU circuit and the QPI IP are shown in Table I, for 4 and 8 assignment/min pipelines. The sizes of the points and the cluster ID buffers are 64 bytes each. Percentages indicate the utilization of a resource. The DSPs are used for the 32-bit single precision floating-point adders, subtractors and comparators.

Similar to Choi and So [15] we use data from the UCI Machine Learning Repository [16]. The data contains over 2 million individual household electric power consumption measurements (some entries are invalid leaving approximately 2 million usable ones). Each measurement has 9 attributes, but similar to Choi and So [15] we use two/four columns to generate 2-dimensional/4-dimensional data.

We implement an optimized software version of K-means in C++. We extend this version to use the AFU and we use it as our baseline for computing the speedup of AFU-assisted execution. This software implementation is multi-threaded, supporting the use of multiple CPU threads. It is compiled with `g++` with optimization level `-O2`.

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 the clustering. The *update time* is the time to preform the update step of K-means.

We use Intel's Performance Counter Monitor API [17] to query the processor's internal counters and measure the L3 cache misses and the number of memory requests to the DRAM controller. We use these values to reason about data sharing between the processor and the AFU.

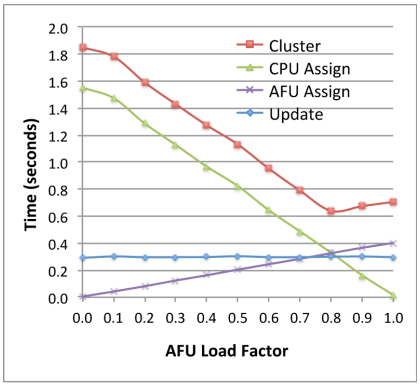

Fig. 2. Clustering as a function of the AFU load factor.

B. Results

We present our results using 8 assignment/min pipelines, clustering 2 million data points in 8 clusters[1].

Figure 2 shows the clustering time, using a single CPU thread plus the AFU, as a function of the AFU load factor. The figure also shows the time to perform the assignment step by the CPU thread and the AFU. It also gives time to update the centroids by the CPU thread. A number of observations can be made from this figure.

First, the assignment step on the AFU by itself (i.e., load factor of 1) is 3.8X faster than on the CPU thread by itself (i.e., load factor of 0). This reflects the performance improvement that is attained by the AFU compared to a single CPU thread.

Second, the overall clustering time initially declines as the AFU load factor is increased. This decline is due to the CPU thread offloading increasing amounts of assignment computations to the AFU, as can be seen from the times to perform the assignment steps by each. This decline continues until the time taken by the AFU to perform the assignment step is roughly equal to that of the CPU thread, at a load factor of about 0.8. Further increasing the load factor causes the AFU to take longer than the CPU thread, leading to an increase in clustering time.

The optimal load factor of 0.8 reflects the fact that the AFU is 3.8X faster than a single thread. Thus, roughly, $(4/5)^{th}$ of the computations should be assigned to the AFU to balance workloads.

The clustering computations at the "optimal" load factor of 0.8, are 2.9X faster than the CPU thread by itself and 1.1X faster than the AFU alone. This shows 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.

Finally, Figure 2 also shows that the time of the update step remains relatively constant, which is expected since the entire step is performed solely by the CPU thread, irrespective of the load factor.

Figure 3 depicts the clustering time as a function of the AFU load factor for different number of CPU threads. The figure indicates a similar trend to that of the single CPU thread execution. Clustering time declines until an optimal load factor

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

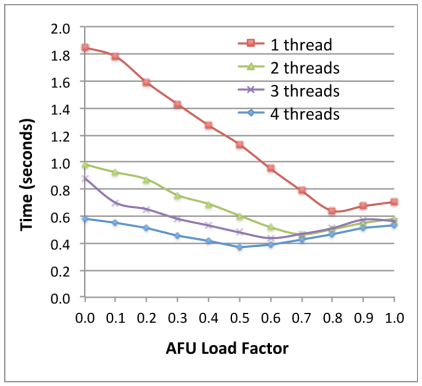

Fig. 3. Clustering time for single- and multi-threaded executions.

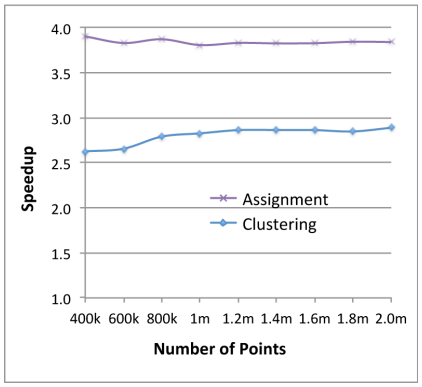

Fig. 4. Speedup of AFU usage with increasing input data size.

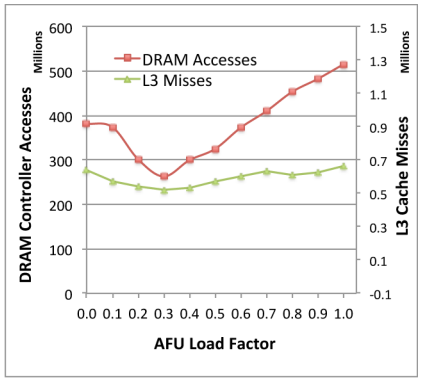

Fig. 5. L3 cache misses and DRAM bytes read/written.

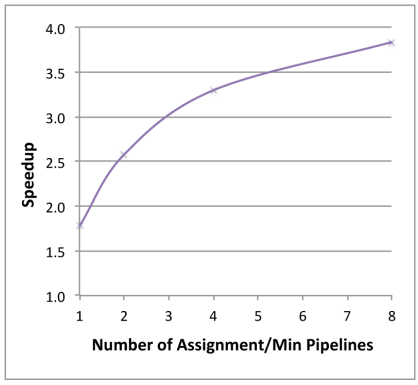

Fig. 6. Speedup of AFU for different number of AFU pipelines.

is reached and then starts to increase. However, the value of the optimal load factor is a function of the number of CPU threads. The more CPU threads are used, the smaller the optimal load factor is. This is because more CPU threads increases the computational power of the processor in relation to the AFU and thus, a smaller load factor balances the workloads.

The improvements in CPU-alone (i.e., load factor 0) performance with increasing number of threads suggests that the optimal load factor can be estimated as $f_{opt} = 1/(1+t/3.8)$, which generally agrees with the optimal load factors seen in Figure 3.

Figure 4 shows the clustering and assignment speedups with a single CPU thread, as a function of the size of the input data. The clustering speedup is at the optimal AFU load factor and is computed with respect to the software-only execution for the respective data sizes. The figure shows that the speedup is largely not sensitive to the size of the input data and will likely be the same for larger data. The small improvement of the clustering speedup at small data sizes is due to the better amortization of overheads.

We use the processor's PCM counters for a first-order analysis of the impact of data sharing between the CPUs and the AFU. Figure 6 gives the number of bytes read from and written to the DRAM memory controllers of the processor (in millions of bytes) during the update step of K-means. It also shows the number of L3 cache misses (in millions of misses) for the same step. Both are shown as a function of the AFU load factor and for a single CPU thread.

The figure shows that as the load factor initially increases, both the L3 misses and the number of DRAM reads/writes decrease. We attribute this to improved cache utilization as the CPU thread processes less points with increasing load factor. However, when the load factor continues to increase, the number of L3 misses increases and the number of DRAM reads/writes sharply rises. We attribute this to the increasing number of accesses the CPU makes to the data written by the AFU during the update step, namely, the `clusterID` array. These misses are more likely to be serviced from memory since the AFU's cache is relatively small (64 KB) compared to the size of the portion of the `clusterID` array written by the AFU (4MiB at a load factor of 50%). This is indeed reflected in the sharp increase in the number of bytes read and written to the DRAM controllers of the processor. Nonetheless, as Figure 2 shows, the update step time remains constant. Thus, while the sharing data during the update step of the computation does increase memory traffic, impact on performance is minimal.

Figure 6 shows the assignment speedup of the AFU over a single CPU thread, as a function of the number of pipelines in the AFU. The speedup increases with the number of pipelines. However, the improvement is not linear and there are diminishing returns. Increasing the number of pipelines from 4 to 8 improves the speedup from 3.3 to only 3.8 with a significant increase in resources, particularly the number of DSP blocks used, as shown in Table I.

Finally, Figure 7 shows the impact of memory accesses, in relation to computations, on the speedup of the AFU. It

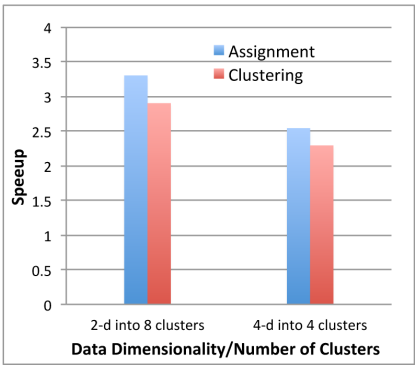

Fig. 7. Speedup of AFU for different data dimensionality and clusters.

shows the clustering and assignment speedups for the AFU with 4 pipelines using 2-dimensional data with 8 clusters and 4-dimensional data with 4 clusters. The amount of computation performed on the AFU in each case is roughly the same (i.e., same number of DSP blocks for floating point operations). However, in the case of 4-dimensional data twice as many cache lines must be fetched from memory by the AFU, reducing speedup.

VI. 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 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 use both to perform the clustering.

Hussain et al. [12], [13] implement K-means completely in hardware, but use fixed-point arithmetic and demands all data be resident in the FPGA's memory blocks. This limits them to smaller data sets of about 3000 points and requires that data be explicitly moved to the memory blocks.

Lavenier [10] 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 [11].

Lin et al. [14] 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 for limited size data sets.

Choi and So [15] accelerate a map-reduce K-means on a cluster of nodes equipped with FPGAs. They add a dedicated communication network for the FPGAs, process large data sets and report speedups of up to 20X over a Java implementation on a three node cluster.

There has been also recent work on exploring shared memory CPU-FPGA systems. Weisz et al.[18] examine pointer chasing using the FPGA fabric and show that it is efficient, especially with processor assistance. Zhang et al. [19] develop high throughput sorting on the same platform we use in this work.

VII. Concluding Remarks

We use K-means as a case study for the design of FPGA accelerators for the Intel QuickAssist FPGA platform, a tightly-coupled shared-memory processor-FPGA systems. The presence of a shared memory and the ability of the accelerator to directly read from and write to memory allowed a design in which data is only logically partitioned (and re-partitioned) between concurrent CPU threads and the FPGA accelerator. This provides an improvement in performance over using only CPU threads or only the FPGA accelerator. Experimental evaluation shows that the combined use of the AFU with a single CPU thread, the improvement is 2.9X over the CPU thread alone and 1.1X over the accelerator alone. With 4 CPU threads, the improvements are 1.6X and 1.9X respectively. Further, increased memory traffic has minimal impact on performance.

There are a number of ways in which this work can be extended. First, it is possible to also accelerate the partial reduce step of K-means on the AFU, allowing both CPU threads and the AFU to perform it. Second, it is possible to approximate the floating point arithmetic carried out by DSP blocks by integer arithmetic, freeing resources and allowing an increase in the number of assignment/min pipelines. 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] R. Wilson, *Heterogeneous Computing Meets the Data Center*, 2014. [Online]. Available: https://www.altera.com/solutions/technology/system-design/articles/_2014/heterogeneous-computing.html
[2] P. Gupta, *Xeon+FPGA Platform for the Data Center*, 2015. [Online]. Available: http://www.ece.cmu.edu/~calcm/carl/doku.php?id=pk_gupta_intel_xeon_fpga_platform_for_the_data_center
[3] B. Wile, *CAPI is Core to POWER*, 2014. [Online]. Available: http://www-03.ibm.com/linux/blogs/capi/
[4] Xilinx Inc., *Zynq-7000: all programmable SoC*, 2014. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
[5] Micron Inc., *Convey CAPI Developer Kit for Xilinx FPGAs*, 2015. [Online]. Available: https://www.micron.com/about/about-the-convey-computer-acquisition/capi-developer-kit
[6] J. Wu, *Advances in K-means clustering*. Springer-Verlag Berlin Heidelberg, 2012.
[7] S. Lloyd, "Least squares quantization in PCM," *Trans. on Info. Theory*, vol. 28, no. 2, pp. 129–137, 1982.
[8] Intel Corp., *Intel QuickAssist Technology*. [Online]. Available: http://www.intel.com/content/www/us/en/embedded/technology/quickassist/overview.html
[9] ——, *Intel QuickPath Interconnect (Intel QPI) FPGA Performance Reference*, 2015.
[10] D. Lavenier, "FPGA implementation of the k-means clustering algorithm for hyperspectral images," *Los Alamos Nat'l Lab, LAUR #00-3079*, pp. 1–18, 2000.
[11] M. Gokhale, J. Frigo, K. Mccabe, J. Theiler, C. Wolinski, and D. Lavenier, "Experience with a hybrid processor: K-means clustering," *J. of Supercomputing*, vol. 26, no. 2, pp. 131–148, 2003.
[12] H. M. Hussain *et al.*, "FPGA implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data," in *Adaptive Hardware and Systems*, 2011, pp. 248–255.
[13] ——, "Highly parameterized k-means clustering on FPGAs: Comparative results with GPPs and GPUs," in *Reconfigurable Computing*, 2011, pp. 475–480.
[14] Z. Lin, C. Lo, and P. Chow, "K-means implementation on FPGA for high-dimensional data using triangle inequality," in *FPL*, 2012, pp. 437–442.
[15] Y.-M. Choi and H. K.-H. So, "Map-reduce processing of k-means algorithm with FPGA-accelerated computer cluster," in *ASAP*, 2014, pp. 9–16.
[16] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml
[17] Intel Corp., *Intel Performance Counter Monitor Documentation*. [Online]. Available: http://intel-pcm-api-documentation.github.io/index.html
[18] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-FPGA systems," in *FPGA*, 2016, pp. 264–273.
[19] C. Zhang, R. Chen, and V. Prasanna, "High throughput large scale sorting on a CPU-FPGA heterogeneous platform," Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, California, Tech. Rep. CENG-2015-10, October 2015.