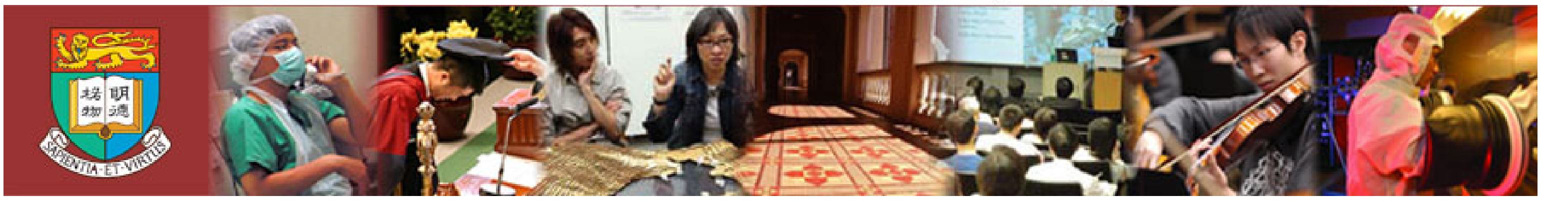


Map-Reduce Processing of K-means Algorithm with FPGA-accelerated Computer Cluster

Yuk-Ming Choi, Hayden Kwok-Hay So
Department of Electrical and Electronic Engineering
The University of Hong Kong, Hong Kong
{ymchoi, hso}@eee.hku.hk

***Abstract*—The design and implementation of the k-means clustering algorithm on an FPGA-accelerated computer cluster is presented. The implementation followed the map-reduce programming model, with both the map and reduce functions executing autonomously to the CPU on multiple FPGAs. A hardware/software framework was developed to manage gateware execution on multiple FPGAs across the cluster. Using this k-means implementation as an example, system-level tradeoff study between computation and I/O performance in the target multi-FPGA execution environment was performed. When compared to a similar software implementation executing over the Hadoop MapReduce framework, 15.5× to 20.6× performance improvement has been achieved across a range of input data sets.**

I. Introduction

The k-means clustering algorithm is a simple yet powerful algorithm that forms the basis of many data analysis applications. It has found applications across many domains including image processing, pattern recognition, machine learning, bioinformatics, data mining and business analytics, etc. Because of its importance and high computational requirements, various acceleration attempts have already been done over the years.

While researchers have demonstrated the performance benefit of implementing the k-means algorithm with FPGAs in various settings, most existing works focused on optimizing the hardware architecture to accelerate their specific application. These solutions worked well on the specific single-FPGA accelerators, but are not readily applicable to problems that demand large-scale cluster computing facilities.

To that end, we present the design and implementation of the k-means clustering algorithm on our multi-FPGA system. Our implementation was designed to follow the map-reduce programming model and was targeted to run on multiple FPGAs installed as accelerators across the cluster. Compared to previous works, our design is unique in that it is general-purpose; it executes across multiple FPGAs, and is readily scalable to larger systems with additional FPGAs.

Using our k-means implementation as an example, this work explores the performance benefits and design considerations of utilizing FPGAs as accelerators in a distributed heterogeneous computer cluster. System-level performance was evaluated and compared against similar software implementations executing on top of the popular Hadoop MapReduce framework. As I/O performance is often the system performance bottleneck, extensive evaluations on inter- and intra-node communication were performed. In addition, by varying the number of available mappers and their distributions among the set of FPGAs in the system, tradeoff between computation and I/O bandwidth was studied.

The remainder of this paper is organized as follows: Section II provides background information and discusses related work. Section III introduces the hardware and software architecture of the k-means implementation. Section IV presents the experimental results from comparing our FPGA k-means implementation against software counterparts. Section V contains system-level performance evaluation on our FPGA system. Section VI concludes this paper with future extension to this work.

II. Background and Related Work

A. *k-means*

The k-means algorithm is one of the most commonly used unsupervised clustering algorithm for data analysis. The goal of the k-means algorithm is to partition the input data into k clusters such that data within a cluster are similar to each other in some way while being dissimilar to data in other clusters. The algorithm proceeds in iterations. Each iteration begins with k centroids corresponding to the k clusters. Each input data object is then assigned to one of the k clusters whose centroid is at minimal distance to the data based on a distance metric, such as its Euclidean or Manhattan distance. Once all data objects are assigned, the centroids of each cluster are updated according to the new partitioning. The algorithm repeats until the centroids remain unchanged at the end of the iteration.

A number of previous works have already demonstrated the benefit of utilizing FPGAs in k-means implementations. Several early attempts have been done to accelerate hyperspectral images clustering using FPGAs. In [7], [10], hardware-software systems combining processor and reconfigurable fabric were proposed, demonstrating over 11× speedup over the corresponding software implementations. Subsequently, the authors in [6], [11] achieved similar speedup in processing multi-spectral and hyper-spectral images by utilizing an FPGA-optimized distance calculation in the k-means algorithm. Likewise, FPGA implementation of k-means has also found useful in real-time image clustering [13].

More recently, in [8], [9], the FPGA implementation of k-means algorithm for processing microarray data was examined and compared to a similar GPU implementation. Furthermore, the authors in [12] explored the use of FPGA-specific hardware structure to accelerate distance computation with high-dimension data in k-means. While these previous works have demonstrated respectable performance, only a single FPGA was employed. Their focuses were to accelerate the particular application using FPGA-specific hardware structure. In contrast, this work examines the use of multiple FPGAs to process large-scale k-means problems by systematically following the map-reduce programming model.

B. Map-Reduce

In its most basic form, map-reduce is a simple programming model that systematically applies an application-specific `map` and `reduce` pure function to the input data list in stages. The `map` function is first applied to each element of the input list to produce an intermediate list, whose elements are subsequently combined by the `reduce` function. As there is no communication between individual instances of `map`, and `reduce` is fully associative and commutative, they offer an enormous opportunity for parallelization.

This powerful programming model is popularized by Google as the MapReduce framework [5], with a now de facto open-source implementation from Apache Foundation called Hadoop [1]. While MapReduce also offers important features such as fault tolerance, the underlying operating principle is similar to the basic map-reduce model. The `map` function takes in a list of key-value pairs and generates an intermediate list of tuples. These intermediate key-value pairs are then processed by different instances of `reduce` function with optional sorting and grouping according to the keys.

As such a promising programming model, map-reduce has also found interest in the FPGA community. To facilitate application development, Yeung et al. proposed a map-reduce library for both GPU and FPGA accelerators [15]. Similarly, focusing on FPGA implementations, Shan et al. presented a MapReduce framework that virtualizes the data synchronization and communication among task scheduler, *mappers* and *reducers* [14]. Both works aimed to improve designers' productivity and portability of the generated MapReduce applications. Our work shares a similar goal of promoting design productivity for large, multi-FPGA map-reduce applications with a focus on CPU-FPGA and inter-FPGA communications.

C. Map-Reduce Processing of k-means

Our implementation of k-means was based loosely on the MapReduce programming model. Each iteration of the k-means algorithm was implemented as a MapReduce job, with the distance calculation implemented as `map` tasks, and the k recentering of centroids implemented as parallel `reduce` tasks.

III. FPGA-Accelerated Cluster Implementation

One key feature of our k-means algorithm is that it runs on multiple FPGAs installed in a computer cluster. Here, we first describe our heterogeneous computer cluster, followed by the design of the FPGA-based MapReduce k-means algorithm and the software support system for such implementation.

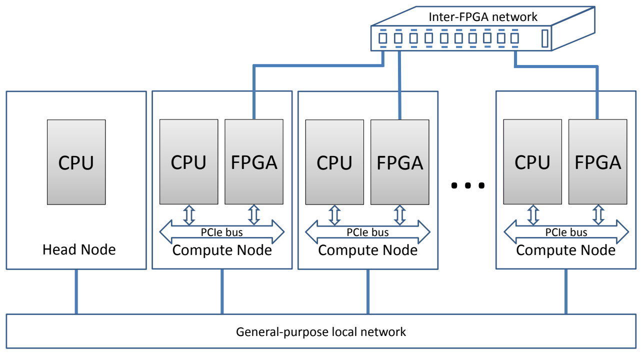


Fig. 1: Cluster overview.

A. Target Computer Cluster

Figure 1 shows an overview of our target heterogeneous computer cluster. Our cluster consists of a homogeneous array of compute nodes. Each compute node is equipped with an FPGA accelerator. On top of basic communication and memory access to the host, the FPGAs are also connected by a dedicated communication network. This simple cluster architecture is scalable and backward compatible with existing systems, therefore enabling us to perform end-to-end performance comparison with existing distributed software systems.

Each compute node is a heterogeneous computer with an FPGA accelerator connected through standard PCIe connection. The host CPU is mainly responsible for managing the attached FPGA. For instance, the host CPU is responsible for configuring the FPGA and marshalling data between the general UNIX file system and the FPGA.

One of the compute nodes is designated either physically or logically as the head node. The main purpose of the head node is to perform job execution and monitoring in the cluster. In the case of our FPGA implementation, a custom management software framework has been developed to manage all FPGA execution from the head node. The node maintains information of compute nodes like IP address and its availability for computation.

Finally, the inter-FPGA network allows direct FPGA-FPGA communication without involving any CPU. In fact, the software system may be completely unaware of this mode of inter-FPGA communication. While such network is essential to provide low latency and high bandwidth inter-FPGA communication, it must also be carefully designed such that it is as scalable as the cluster itself. In our current version, we opt for standard Gigabit Ethernet as the physical implementation of this network as a tradeoff among performance, scalability and reliability.

B. Map-Reduce k-means Implementation

Each iteration of the k-means algorithm is formulated as a map-reduce job in our implementation.

Let N be the number of input data and K be the number of clusters these data are partitioned into. Also, let $C^{(i)} = \{\mu_k^{(i)} : k \in 1, 2, \ldots, K\}$ be the set of center of gravity of the K clusters in iteration i, where $i \in \mathbb{Z}^+$. The set of initial centroids $C^{(0)}$ are K randomly chosen data from the input.

The `map` function takes 1 input data and produces 1 key-value pair where the key is the closest centroid to the input data, and value is the input data itself. To facilitate this computation, it is assumed that the `map` function receives $C^{(i-1)}$ before start of iteration i. In our current implementation, Euclidean distance between the centroid and the data is used.

These intermediate key-value pairs are then grouped by their key in a shuffling stage and are collectively passed to the `reduce` function. The `reduce` function thus takes as input a set of key-value pairs with the same key and computes the new centroid, $\mu_k^{(i)}$, among them.

1) FPGA Computation: Both the `map` and `reduce` functions are implemented using FPGA fabrics. We call each physical implementation of the `map` function in FPGA a *mapper* and that of the `reduce` function a *reducer*.

In theory, with N input data, a maximum of N instances of `map` may be executed in parallel for best performance. In practice, only M physical mappers, where $M \ll N$, are used to timeshare the workload of the N `map` instances. The M mappers may further be implemented across multiple FPGAs.

In our k-means implementation, the number of `reduce` instances required depends on the actual input data. Since K is relatively small in our design, exactly K reducers are used in our design, each of them responsible for computing centroid of one cluster. Currently, all K reducers are physically located within 1 single FPGA. However, multiple FPGAs may be designated as reducer FPGA. Depending on the hashed value of the key, the mapper FPGA groups all the intermediate key-value pairs into different buffers. Key-value pairs stored within a specific buffer are then transferred to a designated reducer FPGA.

Figure 2 shows a high-level block diagram of our k-means design using 3 FPGAs. Within a mapper FPGA, multiple mappers and a central scheduler are presented. At the beginning of each iteration, the starting centroids, $C^{(i-1)}$ are first sent from the host CPU to each on-chip mapper. Each mapper stores these centroids within its own BRAM. Subsequently, input data are continuously streamed from the host CPU to an on-chip input buffer within the scheduler. Each input data is a D-dimensional double-precision floating point number. As soon as a mapper becomes available, it fetches data from this input buffer, computes its Euclidean distance against the stored centroids, and produces a key-value pair accordingly. The key part is an integer representing the closest centroid found and the value part is the data point. Finally, the computed key-value pair is stored in an output buffer, ready to be sent to the reducer FPGA.

On the reducer FPGA, a similar structure can be observed. Once the key-value pairs are received from the dedicated FPGA network, they are processed by the corresponding on-chip reducer module. K reducers are implemented such that each of them may be responsible exclusively for computing the $\mu_k^{(i)}$ of its own cluster. Each reducer has its own set of input and output buffers. It fetches key-value pairs from the input buffer and extracts from it the data point. Subsequently, it accumulates all the received data points and eventually computes $\mu_k^{(i)}$ based on the accumulated value. The newly computed centroid is then stored into the output buffer.

2) Communication: There are two main communication channels in our k-means implementation. The first involves retrieving input data from the general file system to be processed by the mappers; while the other involves passing intermediate key-value pairs between mappers and reducers in multiple FPGAs. The output of the reducers contains K centroid locations and incurs negligible I/O bandwidth.

The input to our k-means algorithm is large data file with up to 100 million data points. In a commercial computer cluster, large data files are normally stored on a distributed file system, such as the Hadoop Distributed File System (HDFS) so multiple processing nodes can process them concurrently. To provide similar facilities to our FPGA design, the input data files are also partitioned into smaller chunks and distributed to the respective hosting nodes of the FPGA accelerators before the algorithm executes.

During the execution of the k-means algorithm, the hardware/software system as shown in Figure 3 is responsible for continuously streaming data from the host to the FPGA. A simple C++ program, called the *Data Manager*, is executed on each compute node whose responsibility is to retrieve data from the partitioned data file in the general-purpose file system. Another software program, called the *Streamer*, interacts with the *Data Manager*. Whenever the *Data Manager* has a batch of input data, the *Streamer* takes over the batch and streams it to the FPGA, with the help of a PCIe driver residing in the Linux Kernel. The PCIe driver copies the batch of data from user-space memory to kernel-space memory. Then, the FPGA is instructed by the driver to perform Direct Memory Access (DMA) read operation on the kernel-space memory. By doing the DMA transfer, the data points are transferred from the main memory to the FPGA through the PCIe channel. Performance of this first communication path is limited by the combined effect of hard-disk speed, OS overhead, as well as PCIe bus transfer speed.

The other major communication facility in the system is the FPGA-to-FPGA communication path between the mappers and the reducers. This communication path takes place through the direct inter-FPGA network as mentioned in Section III-A.

Refer back to Figure 2, the intermediate key-value pairs generated from the mappers are collected by the *Scheduler*. The pairs are then transmitted to the Ethernet core, which packetizes the pairs into Ethernet frames. The MAC address of the reducer FPGA is inserted automatically to the header of each frame by the hardware support system. These frames are then transmitted to the Gigabit Ethernet switch, which routes the frames to the destination FPGA according to the frame's header. The Ethernet core on reducer FPGA de-packetizes the received frames and forwards the key-value pairs in the

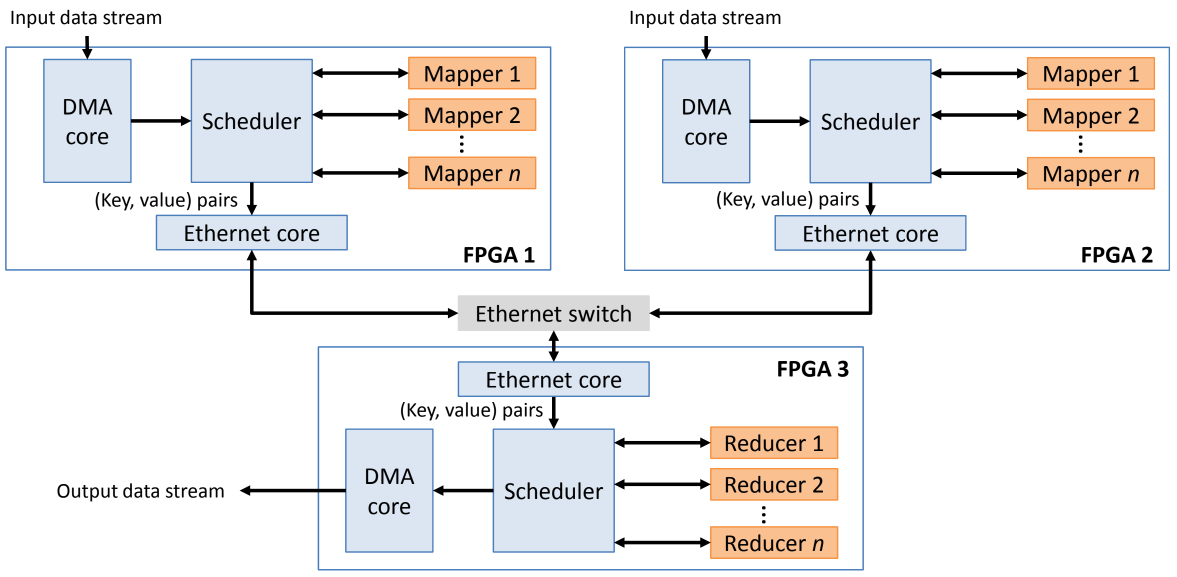


Fig. 2: System architecture block diagram.

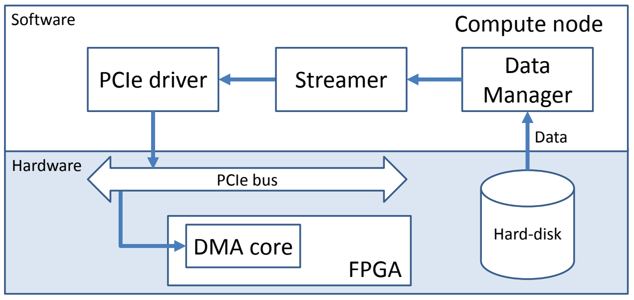


Fig. 3: PC-to-FPGA communication.

payload to the *Scheduler*. Finally, the *Scheduler* looks up the *key* in the received pair and forwards it to the corresponding reducer in the FPGA. Final results are transferred back to the host through PCIe bus.

C. Hardware/Software Management System

To facilitate the execution of hardware/software applications with multiple FPGA accelerators in the cluster, a general-purpose management system has been developed. The system provides resource and process management across the cluster. It also manages the usage of FPGA accelerators among users, and facilitates efficient inter-FPGA communication autonomous to the hosting CPUs.

To run a hardware/software job in the cluster, as illustrated in Figure 4, a top-level program is needed to submit a job request to the head node following a simple server/client model. The job tracker program running at the head node handles the job request by searching a list of available compute nodes and allocating enough nodes for the job. The top-level program is also required to submit the job with a configuration file. The configuration file contains information such as location of the executable of the *Data Manager*, the input data file and so on. After the configuration file is parsed, the executables of both the *Data Manager* and the *Streamer* are copied to the allocated compute nodes and remotely executed using Open MPI library [3]. The *Streamer* process is spawned and instructed to reset the underlying FPGA. Then, input data are retrieved and streamed to the FPGAs in different compute nodes in parallel. When the job is completed, the job tracker carries out some clean-up works such as freeing the used compute nodes.

As such, most of the hardware and software components described in the previous section are general-purpose modules that are reusable. In the PC-to-FPGA communication, the communication channel between the *Data Manager* and the *Streamer* is a software FIFO. Therefore, the *Streamer* can be reused if another application that handles input data differently in the *Data Manager* is to be implemented. Also in the FPGA gateware design, the DMA core and the Ethernet core are designed to be reused for other applications as well. The two cores communicate with the *Scheduler* using FIFO-like buffers. As long as the FIFO-like interface is followed, the logic design of the *Scheduler* is independent of the two cores.

IV. Experimental Results

A. Experimental Setup

The performance of our FPGA k-means implementation was evaluated in the targeted heterogeneous computer cluster. The experiment was run on three compute nodes, each containing a KC705 FPGA board from Xilinx. Each KC705 board contains a Kintex-7 FPGA connected to the CPU through a PCIe $\times 4$ gen. 1 connection. The boards were connected with a dedicated Gigabit Ethernet network. Two of the boards were configured as mappers and one was configured as reducer as shown in Figure 2. The maximum number of mappers $M = 64$ was employed on the two mapper FPGAs, with 32 mappers executing on each FPGA. The final FPGA served as a reducer

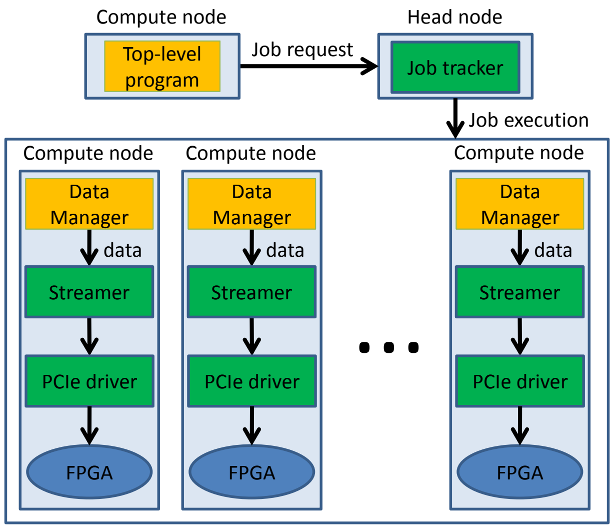


Fig. 4: The cluster management system.

FPGA with K reducer modules implemented. Each FPGA design operated at 250 MHz.

The performance of our FPGA implementation was compared against two software implementations, with one being a baseline implementation and the other being an optimized implementation. The baseline software implementation of the k-means clustering closely followed the original multi-FPGA design so as to provide an one-to-one comparison. The k-means application from the open-source Mahout library [2] version 0.7 was used as the optimized software example. The Mahout library includes a MapReduce implementation of the k-means clustering running on Hadoop cluster and is well-optimized for large input data set. Both software implementations were executed on a Hadoop cluster (version 2.0.0) with 3 compute nodes and 1 head node connected with Gigabit Ethernet. All computer nodes run Linux operating system and are equipped with Intel Core i5 2.90 GHz CPU and 8 GB of RAM.

Input data set to both the hardware and software designs was chosen from the UCI Machine Learning Repository [4]. The data set is an individual household electric power consumption data set. It contains a 4-year record of electric power consumption of a household with sampling rate of one minute. The data set consists of 2075259 entries of 9 attributes. Based on the UCI data set, we created two sets of input data to our k-means implementation. For the first input data set, it consists of 2 attributes extracted from the UCI data set, namely *global_active_power* and *global_reactive_power*. For the second input data set, 4 attributes from the UCI data set are used, which are *global_active_power*, *sub_metering_1*, *sub_metering_2* and *sub_metering_3*. By using the two input data sets in our k-means algorithm, seasonal or monthly power consumption pattern can be observed. Both data sets were stored on compute nodes as a set of D-dimensional data points, with each dimension being a 64-bit double-precision floating point number. As mentioned, the data set used in the multi-

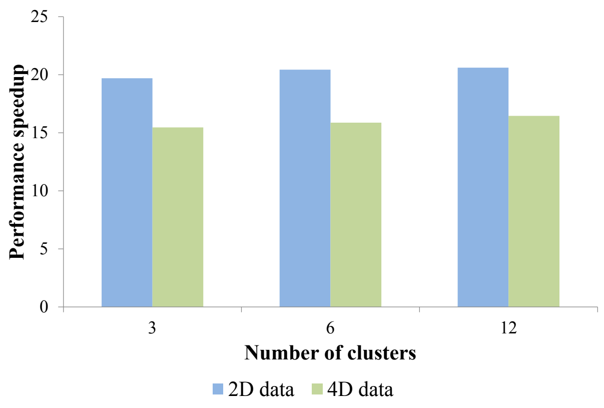


Fig. 5: Performance speedup of the k-means implementation on multiple FPGAs compared to the baseline software version on Hadoop. $M = 64$ in all cases.

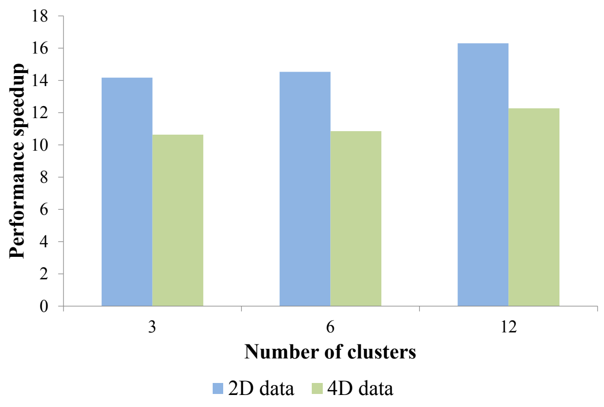


Fig. 6: Performance speedup of the k-means implementation on multiple FPGAs compared to the Mahout version on Hadoop. $M = 64$ in all cases.

FPGA version was manually partitioned into two halves such that the two mapper FPGAs shared the workload from the input data set, while the one in Hadoop version was stored on the Hadoop Distributed File System (HDFS).

In both FPGA and Hadoop versions, the processing time was measured between reading the first byte of data by the mappers and writing the final results to CPU by the reducer.

B. Experimental Results

Figure 5 shows the performance speedup of our FPGA k-means implementation against the baseline software implementation. As shown from the results, the performance of the FPGA implementation is 15.5× to 20.6× faster than its software counterpart. Unfortunately, the speedup advantage of the FPGA implementation over software shrinks as the input data dimension increases. We expect such a small decrease in speedup is due to the increased overhead in streaming data in the FPGA implementation when compared to the software implementation.

Figure 6 presents the results obtained by comparing the

FPGA k-means implementation with the optimized software k-means. The results show that our FPGA implementation can still achieve $10.6\times$ to $16.3\times$ speedup over the optimized software version.

In both software k-means implementations, data are retrieved from the HDFS, copied to main memory and processed by the CPU directly. On the other hand, the FPGA implementation has the additional overhead of copying the entire data set from main memory to FPGA through the PCIe bus. When compared to the software implementation, such additional memory copying overhead only increases as input data size increases, diminishing the speedup advantage of the FPGA implementation.

V. System-Level Performance Evaluation

Despite the fact that up to $20.6\times$ performance speedup is obtained with our k-means implementation on multiple FPGAs, it is important to identify the I/O bottleneck in the system for future improvement. In order to do so, a micro-benchmark designed to measure the I/O performance in the system was carried out with a baseline k-means experiment. The baseline experiment accepted input data sets ranging from $100\,\mathrm{k}$ to $100\,\mathrm{M}$ 2D data points that were randomly generated. The same hardware setup as the previous experiment was used, with 3 KC705 boards as mapper and reducer FPGAs.

The micro-benchmark also involves experiments evaluating the performance of major data paths so that the I/O bottleneck in our system can be pinpointed. The major data paths lie in the host-to-FPGA and inter-FPGA communication channels. The host-to-FPGA communication includes data movement from hard-disk, through main memory and down to FPGA via PCIe bus. The inter-FPGA channel refers to the mapper-reducer communication through the Gigabit Ethernet switch. It is believed that these two channels contribute a large portion of communication overhead to the system.

A. *I/O Performance Evaluation*

Figure 7 shows the throughput performance of the baseline experiment as K varies. By considering the throughput performance, it can be seen that the throughputs of all three values of K stay roughly constant for $M = 64$. In all 3 cases, it is conjectured that the throughput performance is limited by data I/O in the system. On the other hand, the performance for $M = 16$ is more likely limited by the smaller number of mappers. As K increases, the performance is particularly vulnerable to the increased complexity.

To pinpoint the system I/O bottleneck, the following specifically designed experiments were used to measure the throughput performance of the major data paths:

1) Host-to-FPGA test
2) Ethernet line speed test
3) k-means map-only test

The first test aimed to specifically evaluate the performance of the data channel between host PC and FPGA board. Input data were stored on two compute nodes and streamed to the attached FPGA. On the mapper FPGAs, the data streamed from the host were discarded so that the overhead of the k-means algorithm would not be included. The measurement would solely represent the performance of the data channel. Table I shows the throughput performance of the data streaming process. For the case of $100\,\mathrm{M}$ data points, the throughput achieved was 99.44 Megabyte per second (MBps). In other words, the maximum data streaming capacity in the FPGA system was 99.44 MBps.

TABLE I: Processing time of the data streaming process on two mapper FPGAs.

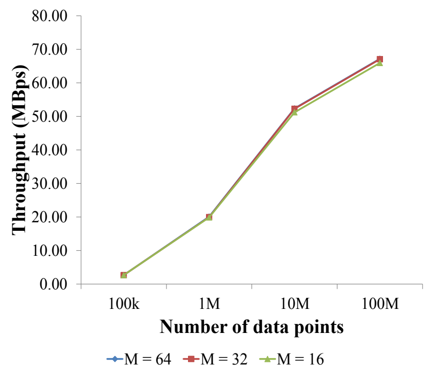
No. of input data points	Processing time	Throughput
$100\,\mathrm{k}$	0.5665 sec	2.69 MBps
$1\,\mathrm{M}$	0.7035 sec	21.69 MBps
$10\,\mathrm{M}$	2.2721 sec	67.16 MBps
$100\,\mathrm{M}$	15.3445 sec	99.44 MBps

The second test was to measure the throughput of the Ethernet channel between mapper and reducer FPGAs. Key-value pairs were artificially generated by the two mapper FPGAs. The generated pairs were then transmitted to the reducer FPGA. The overall transmission time for the key-value pairs in the mapper-reducer communication was measured and shown in Table II. It can be seen that the maximum performance of the mapper-reducer communication was 111.67 MBps.

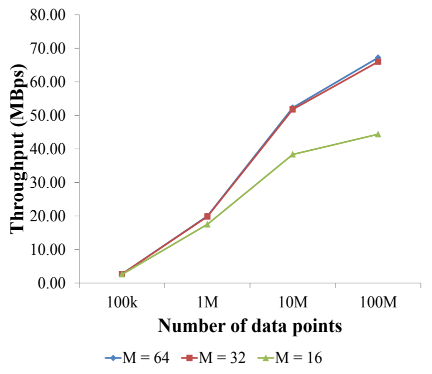
The third test was the same as the baseline experiment, except that the intermediate key-value pairs were discarded at the output of all mapper modules. The key-value pairs were not transmitted to the reducer FPGA for performing the `reduce` function and hence the latency overhead from the Ethernet communication was removed. The overall time for streaming data to FPGA and generating all key-value pairs was measured.

Figure 8 summarizes the results from previous experiments. It shows a model of factors determining the performance of the k-means application. For simplicity, only the case of the largest input data set, $100\,\mathrm{M}$ input vectors, is considered. The theoretical throughput as shown in the figure was measured using the number of hardware cycles required by a mapper to compute the closest centroid in our current implementation.

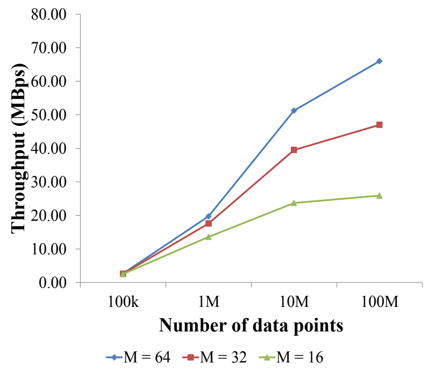
For small number of mappers available, such as $M = 16$, the computational power of the mapper modules is the major limiting factor to the k-means application. This effect is clearly indicated in Figure 8b and Figure 8c, where the three solid lines are very close at $M = 16$, implying that the k-means application performance is heavily limited by the compute capacity of the FPGA system. As the number of mappers increases, the host-to-FPGA communication becomes the major bottleneck to the application performance. For the cases of $M = 32$ and $M = 64$ in Figure 8a, the solid line (k-means map-only) overlaps with the dashed line (Host-to-FPGA channel), pointing out that the k-means performance is bounded by the data streaming capacity. Therefore, the performance of our current implementation of the k-means application cannot be maximized. Immediate next step is accordingly to develop a more efficient implementation on the host-to-FPGA communication.



(a) $K = 3$

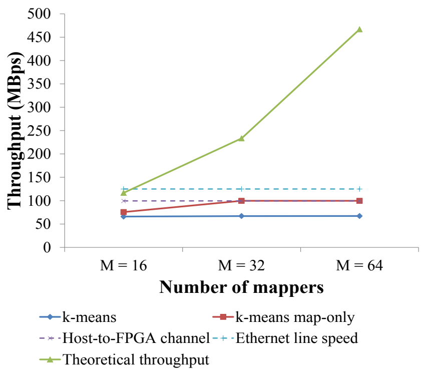


(b) $K = 6$

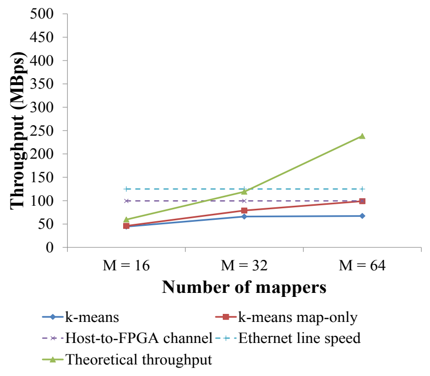


(c) $K = 12$

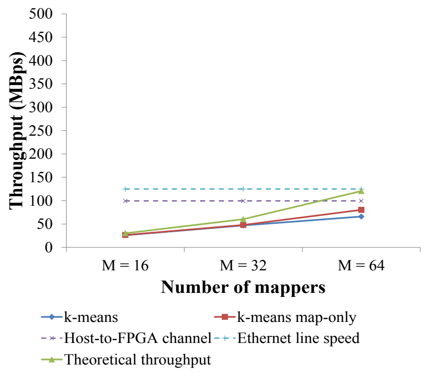
Fig. 7: Effect of varying on K on system throughput performance. $D = 2$ in all cases.



(a) $K = 3$



(b) $K = 6$



(c) $K = 12$

Fig. 8: Performance model of the k-means application. Input = $100\,\mathrm{M}$ points and $D = 2$ in all cases.

TABLE II: Processing time of key-value pairs transmission in mapper-reducer communication.

No. of input data points	Processing time	Throughput
100 k	0.0181 sec	105.50 MBps
1 M	0.1720 sec	110.91 MBps
10 M	1.7091 sec	111.60 MBps
100 M	17.0805 sec	111.67 MBps

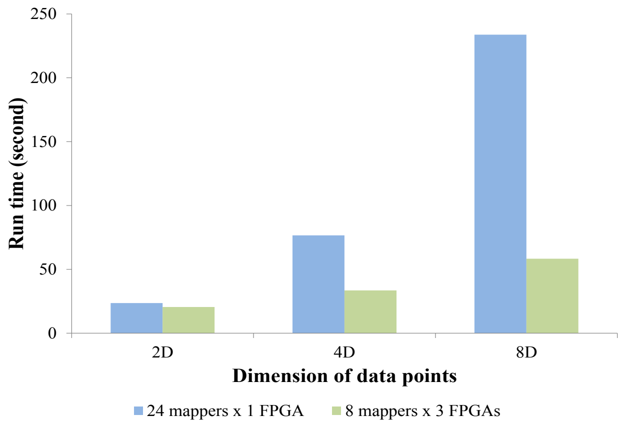


Fig. 9: Performance of FPGA design on variable number of FPGAs. Input = $90\,\mathrm{M}$ points and $K = 3$ in all cases.

B. Effect of Number of FPGAs

Finally, the benefit of utilizing multiple FPGAs to ease the data streaming workload in large data processing is explored. So far, all the experiments in previous subsections have been performed with 2 mapper FPGAs and 1 reducer FPGA. To show the advantage of employing multiple FPGAs, two different system setups were evaluated. In both cases, 24 mappers were employed in the system. However, in the first case, 3 FPGAs each containing 8 mappers were used, while in the second case, only 1 FPGA was used with all 24 mappers implemented. In all experiments, various data dimensions were used, while input data set was fixed at $90\,\mathrm{M}$ points and K remained at 3. The input data set was equally divided into 3 subsets individually stored on the compute nodes. Figure 9 shows their performance results. It is apparent that distributing the same number of mappers across 3 FPGAs consistently outperforms that using 1 FPGA.

TABLE III: Resource consumption of `map` and `reduce` functions on FPGA (Xilinx Kintex-7 XC7K325T). Overall resource consumption for $M = 32$ and $K = 12$.

Modules	Registers	LUTs	DSP48E1s	BRAM
Map	4188 (1%)	4108 (2%)	13 (1%)	72kB (1%)
Reduce	14422 (3%)	13358 (6%)	24 (2%)	36kB (1%)
Overall Map	147343 (36%)	140654 (69%)	416 (49%)	5328kB (33%)
Overall Reduce	179554 (44%)	145159 (71%)	288 (34%)	3960kB (24%)

We attribute the performance benefit of the multi-FPGA implementation to the reduced I/O bandwidth requirement on the mapper FPGA as we split the set of mappers into multiple FPGAs. This effect is particularly prominent with large D. Consider each pair of columns. With 90M 4D data points, the single-mapper-FPGA version is more than 2 times slower than the multi-mapper-FPGA version. With 90M 8D data points, the performance of using only one mapper FPGA is about 4 times slower.

Further experiments may be done in the future so as to better understand the balanced ratio between mappers and FPGAs.

C. Resource Consumption

Table III summarizes the resource consumption of the FPGA k-means design. The modules Map and Reduce show the resource utilization of individual `map` and `reduce` function. The Overall Map and Overall Reduce modules indicate resource usage of the largest `map` and `reduce` designs, which were implemented with $M = 32$ and $K = 12$ respectively.

VI. Conclusions

In this paper, we have presented an implementation of the k-means algorithm using multiple FPGAs in a heterogeneous computer cluster. The implementation took advantage of the map-reduce programming model to allow easy scaling and parallelization across the distributed computer system. A dedicated inter-FPGA communication channel was built in order to allow an efficient and autonomous data movement between FPGAs. A cluster management system was also developed to handle job requests and monitor the overall cluster. Performance evaluations using real-life statistics as input data were carried out and the experimental results were compared with two software k-means solutions running on Hadoop cluster. Our performance results show that the multi-FPGA implementation can outperform the baseline software implementation in all test cases, offering speedup from 15.5× to 20.6×. In addition, the performance of the multi-FPGA implementation was also compared with that of the optimized software implementation offered by the Mahout library. The results demonstrate performance speedup of the multi-FPGA design, ranging from 10.6× to 16.3×. I/O bottleneck analysis was conducted with several specifically designed experiments. It was found that the primary source of I/O limitation within the k-means application was the host-to-FPGA communication channel. Various studies show that, by taking advantage of multiple FPGAs, the I/O communication overhead can be relieved and greater performance improvement can be achieved.

In the future, we plan to increase the scale of the experiment to evaluate the case for utilizing multiple FPGAs in processing large data sets in data centers. Further experiments are planned to better understand the tradeoff between I/O and computational performance limitations.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong, project ECS 720012E.

References

[1] "Apache Hadoop, http://hadoop.apache.org/."

[2] "Apache Mahout, http://mahout.apache.org/."

[3] "Open MPI, http://www.open-mpi.org/."

[4] K. Bache and M. Lichman, "UCI Machine Learning Repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[6] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic Transformations in the Implementation of K-means Clustering on Recongurable Hardware," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, ser. FPGA '01. ACM, 2001, pp. 103–110.

[7] M. Gokhale, J. Frigo, K. Mccabe, J. Theiler, C. Wolinski, and D. Lavenier, "Experience with a Hybrid Processor: K-Means Clustering," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 131–148, 2003.

[8] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "FPGA Implementation of K-means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data," *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 248–255, Jun. 2011.

[9] H. Hussain, K. Benkrid, A. Erdogan, and H. Seker, "Highly Parameterized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 2011, pp. 475–480.

[10] D. Lavenier, "FPGA Implementation of the K-Means Clustering Algorithm for Hyperspectral Images," *Los Alamos National Laboratory LAUR*, pp. 00–3079, 2000.

[11] M. Leeser, J. Theiler, M. Estlick, and J. Szymanski, "Design Tradeoffs in a Hardware Implementation of the K-means Clustering Algorithm," in *Proceedings of the 2000 IEEE Sensor Array and Multichannel Signal Processing Workshop*, 2000, pp. 520–524.

[12] Z. Lin, C. Lo, and P. Chow, "K-Means Implementation on FPGA for High-Dimensional Data Using Triangle Inequality," in *International Conference on Field Programmable Logic and Applications*, ser. FPL '12, Aug. 2012, pp. 437–442.

[13] T. Saegusa and T. Maruyama, "An FPGA Implementation of K-Means Clustering for Color Images Based on Kd-Tree," in *International Conference on Field Programmable Logic and Applications*, ser. FPL '06, Aug. 2006, pp. 1–6.

[14] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10, 2010, pp. 93–102.

[15] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, "Map-reduce as a Programming Model for Custom Computing Machines," in *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '08, 2008, pp. 149–159.