Network-Attached FPGAs for Data Center Applications

Jagath Weerasinghe, Raphael Polig, Francois Abel, Christoph Hagleitner
IBM Research - Zurich
Säumerstrasse 4
8803 Rüschlikon, Switzerland
Email: {wee,pol,fab,hle}@zurich.ibm.com

FPT2016

***Abstract*—FPGAs (Field Programmable Gate Arrays) are making their way into data centers (DC). They are used as accelerators to boost the compute power of individual server nodes and to improve the overall power efficiency. However, this approach limits the number of FPGAs per node and hinders the acceleration of large-scale distributed applications.**

We propose a system architecture to deploy large-scale DC applications on standalone FPGAs, independently of the number of CPUs. In our architecture, the FPGAs are directly attached to the DC network, making the FPGA infrastructure scalable and flexible. This FPGA infrastructure enables the creation of flexible multi-FPGA fabrics by connecting several FPGAs together over the DC network.

We implemented a prototype of the network-attached FPGA and ported a distributed text-analytics application onto such a multi-FPGA fabric. We compared our approach with a SW-only implementation and an implementation accelerated with PCIe-attached FPGAs. The results show that the network-attached FGPAs outperform both other implementations by large margins.

***Keywords*—network-attached FPGA; multi-FPGA fabric, distributed computing, data center.**

I. Introduction

An increasing number of big data analytics applications are moving to the cloud to benefit from the flexible and scalable infrastructure offered by modern DCs. In the mean time, the rapid and continuous growth of the datasets processed by these applications have put a tremendous pressure on the DC computing infrastructure, which struggles to produce the desired results in a timely manner. FPGA is one potential solution to bridge this widening gap because of its known power efficiency and superior communication capacity. We have already started to see this happening in the DC space [1] [2] [3]. In all these attempts, the FPGAs are used as accelerators on PCIe plug-in cards to boost the compute power of individual server nodes and to improve the application response time, power efficiency, or both. However, this type of bus-attached accelerator has two major drawbacks: (1) the small number of FPGAs per node limits the scalability required for the acceleration of large-scale applications, and (2) the dominating static power of the server nodes weakens the full FPGA power efficiency that can be exploited.

Rather than bus-attachment, Weerasinghe *et al.* [4] proposed to deploy standalone network-attached FPGAs in the DCs, aiming to accelerate large-scale applications by interconnecting several FPGAs into a flexible software-defined multi-FPGA fabric. We expect such new DC compute infrastructures to successfully bridge the widening performance gap between applications and servers. The implications of the integration of network-attached FPGAs into a DC infrastructure range from the design of the compute nodes up to the racks, and have been outlined in [4]. In this work, we focus on the architecture and prototype implementation of a suitable network-attached FPGA, and show its potentials in distributed computing in DCs. Our contributions in this paper are threefold:

1) We propose an architecture for running distributed DC applications on FPGAs.
2) We implement a prototype of the architecture on a merchant FPGA.
3) We port a distributed text-analytics application onto the prototype implemented and compare the results with a SW-only implementation and an implementation accelerated with PCIe-attached FPGAs.

The remainder of this paper is organized as follows: We explain the motivation behind this work in Section II, followed by related work on deploying FPGAs in DCs in Section III. We then propose an architecture for network-attached FPGAs and its prototype implementation (Section IV). Next, in Section V, we explain how we build a multi-FPGA fabric using the network-attached FPGA prototype implemented. After that, we explain the application that we port onto such a multi-FPGA fabric (Section VI). We present and discuss the experimental results in Section VII and conclude in Section VIII.

II. Motivation

Big data platforms analyze heterogeneous datasets by sorting, indexing, ranking or clustering. While the amount of data to be analyzed increases continuously, the acceptable time to produce the desired results is shrinking. Timely analysis of big data has a significant impact on productivity and for improving the user experience, eventually leading to growth revenue. To execute analytics tasks in a timely manner over large datasets, big data platforms have started to scale out. For an example, large-scale analytics applications use distributed frameworks, such as Hadoop [5], Spark [6], Dryad [7], and UIMA [8] spread over a cluster of servers in DCs. However, many distributed applications do not scale well on these infrastructures because of (i) the limited compute power of individual server nodes, (ii) the performance of the network

that interconnects them, and (iii) the varying response times. Therefore, to host large-scale distributed applications, those three issues have to resolved.

1) Compute Power: The obvious way to increase the compute capacity for running large-scale applications is to add more servers to the DC. However, server expansion is usually hindered by the DC's power and cooling capacity. One way to increase the power efficiency of individual server nodes is to use accelerators, but unfortunately, the potential boost that an accelerator, which is hosted by a server, can provide is weakened by the dominating static power of server nodes [9].

2) Network Performance: Even if the available compute power is sufficient, inefficient networks hamper the scalability of IO-bound applications. For example, when the number of servers is increased beyond a certain threshold, the training of a distributed deep neural network becomes slow, as the network overhead starts to dominate [10]. Similarly, the scalability of Hadoop TeraSort [11] [12] is hindered by the network throughput performance, particularly in the data-shuffling phase.

Distributed applications, such as search, online shopping, social networking, and high frequency trading are interactive in nature with stringent latency requirements. When these applications are running in DCs, a major cause of poor network latency is the SW-based packet processing in the server nodes. The OS network stack takes around 15μs to take a network packet from the wire to the application, where as modern network switches and network interface cards take only around 1μs.

3) Varying Response Time: In synchronous cluster applications, a typical cause of degraded performance is variance in processing times across different servers, leading to many servers waiting for the single slowest server to finish a given phase of computation [10]. This variance in processing times is caused by unpredictable scheduling of CPU and IO resources. To alleviate these issues, some distributed applications use straggler mitigation methods by cloning the same task multiple times [13], which, however, does not solve the fundamental issue.

In summary, the scaling demand of modern DC applications in terms of compute and communication capacity is exceeding the historical scaling rate of the server performance. This creates an increasing mismatch between the resources available, calling for novel big data processing technologies.

III. Related Work

The common approach for deploying FPGAs in a server is by tightly coupling one or two FPGAs to the CPU over the PCIe bus. However, this PCIe attachment has two major issues in DC deployment. First, the power consumption of a server is order of magnitude higher than that of an FPGA. Hence, the power efficiency that can be gained by offloading tasks from the server to 1 or 2 FPGAs is very limited [9]. Second, in DCs, the workloads are heterogeneous and run at different scales. Therefore, the scalability and the flexibility of the FPGA infrastructure are vital to meet the dynamic processing demands. With PCIe attachment, a large number of FPGAs cannot be assigned to run a workload independently of the number of CPUs, and also those FPGAs cannot be connected in flexible user-defined topologies. Some large-scale FPGA deployments [1] get around this issue of scalability and flexibility to a certain extent by having a secondary dedicated network connecting multiple PCIe-attached FPGAs together. However, a dedicated secondary network breaks the homogeneity of the DC network, and increases the infrastructure management overhead.

The other approach for deploying FPGAs is by attaching them directly over the DC network [2] [3] [14], which significantly improves the scalability and the flexibility of the FPGA infrastructure compared with the PCIe attachment. Even though those attempts provide a network connection, the FPGA always remains physically attached, hosted and controlled by a dedicated server. Instead, [4] proposed the concept of standalone network-attached FPGA to completely disaggregate the FPGA resource from the server. This approach frees the FPGA from the traditional CPU-FPGA attachment and tightly couples the network and application processing in the same FPGA device. We believe that this is the key enabler for large-scale deployments of FPGAs in DCs.

The network attachment opens new opportunities, such as connecting multiple FPGAs together in flexible topologies in a DC according to the application demands. For example, several FPGAs can be configured into a multi-FPGA fabric in the form of a pipeline or a tree. However, forming flexible multi-FPGA fabrics using network-attached FPGAs in DCs and porting distributed applications to such fabrics have not yet been addressed. In this work, we focus on porting a distributed application onto such a multi-FPGA fabric and evaluate its performance.

IV. Network-Attached FPGA

This section explains the architecture of the network-attached FPGA and the implementation of the prototype. As shown in Figure 1, the network-attached FPGA is partitioned into two main layers: (i) the application layer, which we call virtual FPGA (vFPGA), and (ii) the network service layer (NSL). Aiming to provide line-rate (10G) network connections to the applications running in the FPGA, these two layers are implemented with a 64 b bus at 156.25 MHz. To speed up the development time, we used Xilinx Vivado High Level Synthesis (HLS) to implement the prototype.

A. vFPGA

The vFPGA hosts the application. The vFPGA has one or more communication links through the NSL to the servers and to other vFPGAs over the DC network. As shown in Figure 1, we call these communication links the data path of the network-attached FPGA. The links can be reliable connection-oriented, such as TCP, or unreliable connection-less, such as UDP. These communication links offered to the vFPGAs are FIFOs, relieving the user (application writer) from complex network programming tasks. The data fed to the TX FIFOs

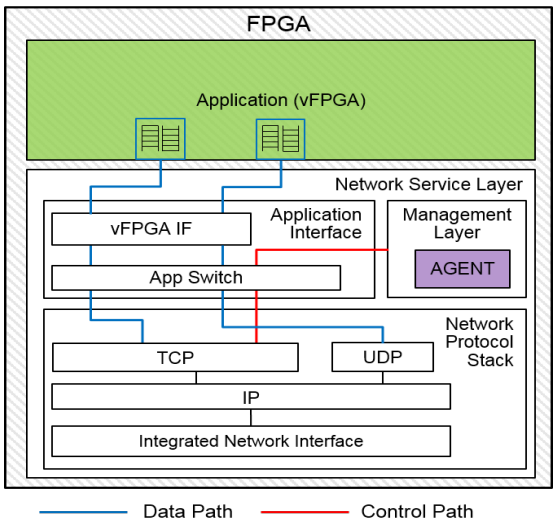

Fig. 1. Network-attached FPGA prototype

is wrapped in network packets and forwarded to the relevant destination by the underlying NSL. Similarly, the network packets received by the NSL are first unwrapped, and then the payload is fed to the RX FIFO to be used by the application. The user distinguishes the various communication links to the vFPGA from each other using the FIFO index. When application-specific protocols are needed, they can be built within the vFPGA atop the FIFO-based TCP or UDP links. The use of the vFPGA in a real application is discussed in the Section VI.

B. Network Service Layer (NSL)

The NSL provides the network connection for vFPGAs to communicate with servers and other vFPGAs over the DC network, similarly to the power service layer (PSL) of IBM's coherent accelerator processor interface (CAPI), which provides a PCIe-based communication link for its accelerator function units for communicating with SW applications. The NSL consists of (i) an application interface layer, (ii) a management layer, and (iii) a network protocol stack.

1) Management Layer (MGL): The MGL includes one or more agents that listen on a pre-defined TCP port for commands and management data from an external SW service, which we call FPGA management utility (FMU). The communication link to the MGL is called the control path of the network-attached FPGA. The agents may include functions, such as configuring a new vFPGA using partial reconfiguration[1], vFPGA monitoring, and executing other utility functions. In this implementation, the management layer runs an agent that can execute TCP listen, connect, and close commands on the underlying FPGA network protocol stack to connect multiple FPGAs together in a software-defined manner.

[1]In this work, we do not implement partial reconfiguration and hence it is not further discussed.

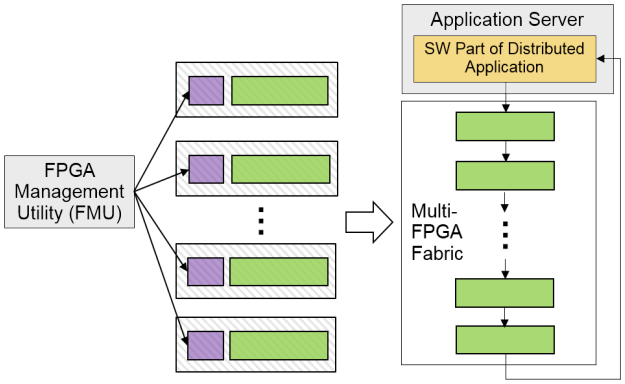

Fig. 2. Software-defined multi-FPGA fabric

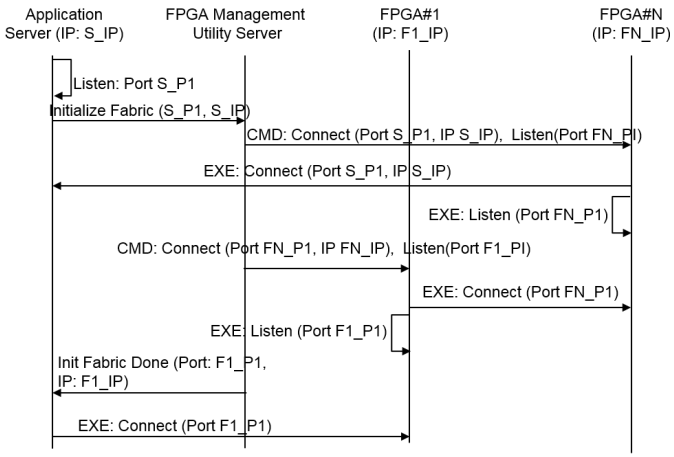

Fig. 3. The flow of forming a multi-FPGA fabric

2) Application Interface Layer (AIL): The AIL executes two main tasks: (i) It servers as a switch that multiplexes and de-multiplexes incoming and outgoing data path and conrol path network payloads to and from vFPGAs and MGL. (ii) It offers multiple FIFO-based TCP and UDP network interfaces to the vFPGAs through the vFPGA IF. The AIL can have one or more vFPGA IFs, so that the network-attached FPGA can run multiple applications simultaneously. The AIL also offers a FIFO-based TCP interface to the MGL.

3) Network Protocol Stack (NPS): The NPS contains a network interface and a protocol stack to connect the FPGA to the DC network. The protocol stack may contain any network (L3) or transport (L4) protocol, which runs atop Ethernet. In our implementation, the protocol stack contains a TCP/IP stack from Xilinx, which is also developed in vivado HLS. The NPS executes all the networking tasks, such as connection management, and only forwards the payload of the network packets to the AIL.

V. Multi-FPGA Fabric

Usually, when several FPGAs are required for an application, either those FPGAs are soldered on a PCB [15] [16] [17] [18] or connected in a network on a fixed topology [19] [1]. However, these implementations are not flexible enough

to be deployed in general-purpose DCs, as the applications running on those DCs are heterogeneous and run at different scales, creating ever changing requirements for the underlying compute infrastructure. In general-purpose DCs, several FPGAs must be able to be connected in fexible topologies on demand and to be released when no longer needed.

Once the network-attached FPGA presented in Section IV is up and running in a DC, several such FPGAs can be connected together over the DC network flexibly in a software-defined maner according to the application demands, as shown in Figure 2. We call this configuration of multiple FPGAs a software-defined multi-FPGA fabric (SDMFF). In this section, creation of such an SDMFF in a pipelined manner is explained.

To form an SDMFF, the FMU introduced in Section IV is used. The server running the SW part of the distributed application first starts listening on a particular port. Then it sends a request to the FMU with its listening port (S_P1) and IP address (S_IP). Upon receiving the request from the server, the FMU sends connect and listen commands to each FPGA as shown in Figure 3. Accordingly, each FPGA executes the commands received by the FMU to connect with the server application and the other FPGAs. Once all the FPGAs are connected, the FMU sends a reply to the server with the listening port (F1_P1) and the IP address (F1_IP) of the first FPGA in the pipeline. Once this information has been received, the server starts a connection to that listening port (F1_P1). With that step completed, the SDMFF has been formed and the distributed application is ready to start execution.

VI. Application

We ported a distributed text-analytics application to the SDMFF described in Section V. In this section, we explain that text-analytics application, the distributed computing framework on which the application runs, and how we integrate network-attached FPGAs into this framework.

A. Text Analytics

Text analytics refers to the task of information extraction from documents containing natural-language text. The main aim is to transform the unstructured information contained in these documents into a structured form, i.e. tables. This is an important processing step in many of today's analytics applications ranging from social media analysis to compliance check and data-center log surveillance. The document-level analysis needs to be run before any higher-level algorithms can perform further analyses.

Text analytics involves several steps from the natural-language processing (NLP) domain, such as tokenization or named-entity recognition (NER). Each step may be performed by either a rule-based or a machine-learning-based implementation. While machine-learning-based approaches are well established, rule-based approaches are often maintained in the enterprise domain to achieve fast and deterministic results [20].

Tokenization is usually the first step when analyzing a document, which breaks up the input text into individual words and characters. For many Western languages, whitespace tokenization is often sufficient to produce a useful set of tokens by splitting the text on whitespace and punctuation characters. The tokens are then used by subsequent processing steps such as dictionary pattern matching or distance checks (how many tokens are two entities apart).

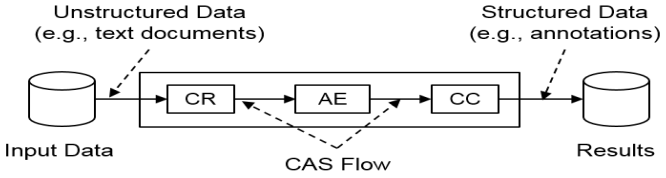

Fig. 4. UIMA pipeline

Named-entity recognition (NER) identifies words or patterns in the document text and assigns them to categories. Words or word sequences can be determined to be a person's name or a geographic location, whereas character sequences may be identified as telephone numbers or date/time stamps. Thus NER involves pattern matching in the form of dictionary matching, which requires a pattern to match on token boundaries, and regular expressions, which operate on the document text without token definitions. Both operations are essential for text analytics and have shown significant performance benefits when run on FPGAs [21] [22].

B. UIMA

Several libraries exist to perform natural-language processing tasks. Every library uses its own type system to create and exchange information between different processing steps. This makes the integration of several libraries into a single application difficult. The Unstructured Information Management Applications (UIMA) [8] specification is an OASIS standard that defines how various components can define a common type system and how data is exchanged between them. The UIMA framework also manages the execution of multiple components and can be configured to create a processing pipeline.

UIMA defines multiple component types and the Common Analysis Structure (CAS) data structure. The CAS object is the central data structure that is created for every document. Every analysis step can retrieve information from the CAS or create results on it. As shown in Figure 4, the three main components to create a UIMA processing pipeline are (i) Collection Reader (CR), (ii) Analysis Engine (AE), and (iii) CAS Consumer (CC).

The CR is the input component and retrieves the documents from an input source, i.e., filesystem or database. It creates the initial CAS object and passes it to the actual processing pipeline. The processing pipeline is a so-called aggregated analysis engine (AE) consisting of multiple primitive AEs. The UIMA framework will call specific routines from each of the primitive AEs to process every document. When processing completes, the CS will receive the CAS and store the relevant results to an output destination.

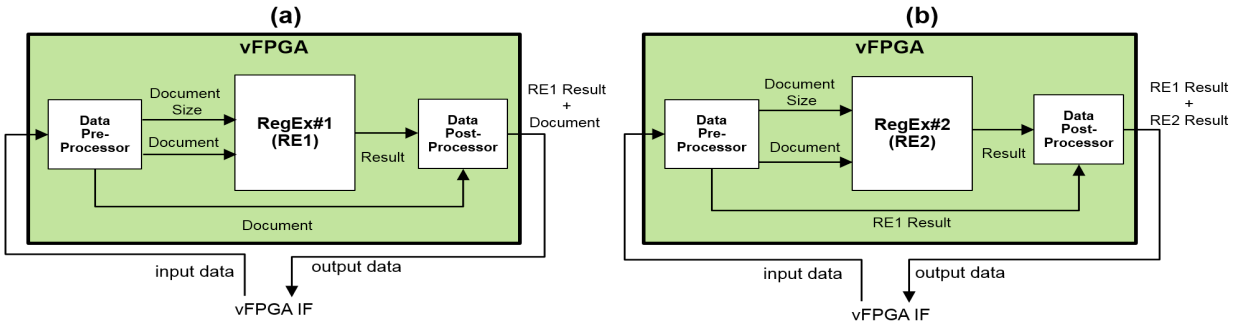

Fig. 5. Implementation of vFPGA for the application: (a) network-attached FPGA1 and (b) network-attached FPGA2

To create large-scale high-throughput and low-latency applications, UIMA offers an asynchronous scaleout (UIMA-AS) [23] version. Instead of calling the processing routines of the AEs synchronously, the individual AEs receive their input CAS from a queue that is assigned to each AE. This allows every AE to work on its own input queue whenever data becomes available. It also simplifies multi-threading as multiple individual threads that run the same AE share the same input queue.

To scale out the analysis application to multiple machines in a cluster, UIMA-AS uses the Java Message Service (JMS) and a messaging broker to manage the queues. The CAS object is communicated via the individual nodes as a serialized XML object, and connections are made through TCP or HTTP. If the CAS object remains on a node, it is kept as a binary object. On each node, one or multiple AEs can be deployed together with their corresponding queues. The deployment descriptor of UIMA-AS specifies how many threads should be used to run a specific AE and additional parameters. The CAS object is then sent around between the individual nodes for processing before returning to the master node, where the application resides.

When the FPGA is attached via PCIe to each node, there is no change in the communication scheme, but the AE's code that uses the FPGA needs to be adapted. For using network-attached FPGAs, we alter the communication structure by adding a sending (TX AE) and a receiving (RX AE) primitive AE to the processing pipeline that remain on the master node. The descriptor of the TX AE contains all information necessary to set up the multi-FPGA fabric pipeline before starting the processing step. In this setup, the FMU is also running in the master node. Once the fabric is formed, every FPGA knows where it receives data from and where it has to send its results. When processing, the TX AE will send only the text document embedded in the CAS object to the first FPGA in the processing pipeline and forward the CAS to the RX AE. As this is a local operation, no network communication is involved. The RX AE will wait for the FPGA processing pipeline to complete processing the text document, which was embedded in the CAS, and then add the results to the actual CAS object.

This setup allows a flexible and efficient communication

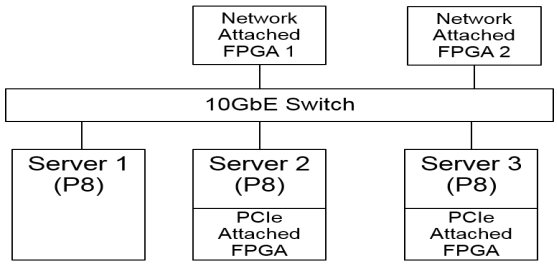

Fig. 6. Experimental setup

without implementing a complex JMS stack on the individual FPGAs. The UIMA-AS framework enables the TX and RX AEs to continuously send and receive data while maintaining UIMA compliance.

C. Text Analytics on FPGAs

To run a text-analytics task on the FPGA, we use IP cores generated by the compilation framework presented by Polig *et al.* [24]. The framework compiles queries written in the Annotation Query Language (AQL) to a hardware description (verilog) that can be synthesized to FPGA logic. The top-level module generated uses AXI-Streaming-like interfaces to accept an input document and produce output results. The input document stream is synchronized with an optional token stream, which defines the token boundaries. If not token definitions are available, this stream must indicate the document size for the input logic to consume the document stream. The results are annotations in the form of four integers: two defining the begin and end the position of the annotation, and one identifier value to determine the type of annotation.

To employ generally available libraries, we use the compilation framework only to compile regular expressions. Regular expressions can be implemented in Java using the built-in java.util.regex package. This enables a straight forward software reference implementation for a UIMA processing pipeline.

D. Implementation

In this work, we use a regular expression IP core as the application, and an SDMFF that consists of two network-

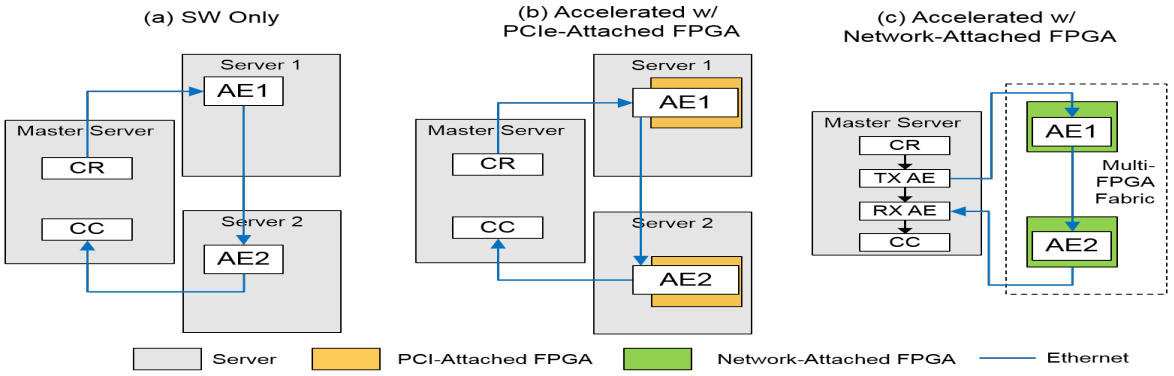

Fig. 7. UIMA pipeline-based experimental cases

attached FPGAs. This application is hosted by the vFPGA as shown in Figure 5. The vFPGA has three main components: (a) a data pre-processor, (b) a data post-processor, and (c) the regex core. The data pre-processor and the post-processor are collectively called the vFPGA application wrapper. The data pre-processor executes two functions: First, the regex core must be fed with each document and its size. When documents are sent over TCP, the document boundaries are lost. Hence, the document boundaries must be recovered before feeding data to the application. Therefore, a messaging layer atop TCP is implemented in the data pre-processor. The server sends each document with its size prepended in 8 bytes. According to the document size, the messaging layer stacks TCP data belongs to each document into a single AXI stream message and forwards it to the application. Second, as the data bus width of the regex core is 128 bit, but the bus width of the vFPGA is 64 b, we convert the bus width from 64 b to 128 b. For the conversion, we used Xilinx AXI stream data-width converters.

The data post-processor first converts the data width of the results generated by the application from 128 bit to 64 bit, and then combines the converted result with the data coming directly from the data pre-processor before forwarding everything to the vFGPA interface.

As shown in Figure 5, the data pre-processor of vFPGA in network-attached FPGA1 forwards the document as it is to the data post-processor, whereas in the vFPGA of network-attached FPGA2, the data pre-processor discards the document after feeding the application with the document data and its size. After that, the data pre-processor forwards the result received from the vFPGA of network-attached vFPGA1 to the data post-processor. The data post-processor of the vFPGA in network-attached FPGA2 combines the results of network-attached FPGA 1 and 2, and forwards it to the vFPGA IF, which eventually ends up in the RX AE in the UIMA framework.

VII. Experiments, Results and Discussion

We evaluated our architecture in terms of network latency, throughput, and lantecy variation. The network-attached FPGA architecture presented was implemented and validated on a Alpha Data PCIe card featuring a Xilinx Virtex7 XC7VX690T FPGA. The design uses one 10GbE network interface.

A. Experiments

As shown in Figure 6, the experimental setup consists of up to three server nodes, each equipped with two IBM POWER8 processors running at 3.5 GHz. Two servers contain commercial off-the-shelf FPGA accelerator cards using an Altera Stratix V A7 FPGA and are connected to the processor via CAPI. All servers run Linux Kernel 3.10 and use IBM Java version 8 and UIMA 2.8.1. The network-attached FPGA cards are plugged into a PCIe expansion chassis [25], from which only power is taken for the cards. The FPGA cards and the three servers are connected to a 10 GbE top-of-rack switch.

We considered three experimental cases: (i) a SW-only implementation (Figure 7-(a)), (ii) an implementation accelerated with PCIe-attached FPGAs (Figure 7-(b)), and an implementation accelerated with network-attached FPGAs (Figure 7-(c)). One server is considered the master server and is responsible for the running the collection reader (CR) and the CAS consumer (CC). It also runs the aggregate analysis engine, which coordinates the order in which the primitive analysis engines (AEs) are executed. The primitive AEs are run by the other two server nodes. These run either the software-only variants of the AEs implemented in pure Java or the FPGA-accelerated version.

For the network-attached scenario, the two servers running the primitive AEs are replaced by the network-attached FPGAs. The master server remains in charge of reading the documents and collecting the results, but now also sets up the multi-FPGA fabric using the FPGA management utility described in Section V, and sends and receives data to and from the FPGAs. Internally the CAS object moves from the sending AE (TX AE) to the receiving AE (RX AE), which involves no data movement or processing.

As explained earlier, both of our AEs perform regular expression matching. AE1 identifies several date formats, while AE2 reports credit card numbers in the document. The SW case uses the standard Java regular expression class, whereas the FPGA version is compiled using [24].

B. Results

The latency and the throughput measurements are done by using standard UIMA tooling. By running $runRemoteAsyncAE$, the number of processed documents and characters are reported together with the required run time. These numbers represent an end-to-end measurement for processing a document collection. The scaleout deployment has been tuned by using available knobs from the UIMA-AS framework, such as the number of threads running an AE or taking care of the (de-)serialization of the CAS, CAS pool size and memory requirements.

1) Latency: Figure 8 shows the latency results in milliseconds for different document sizes. The SW-only implementation took 6, 8, and 20 ms for the document sizes of 512, 1024, and 2055 B, respectively, whereas the implementation accelerated with PCIe-attached FPGAs took 6, 7, and 18 ms. Regardless of the document size, the network-attached FPGA implementation took 0.5 ms. When moving from SW-only version to PCIe-FPGA version, we observed a minor improvement in latency. In contrast, for network-attached FPGAs, the latency improved by a factor of 40 compared with SW-only version.

2) Throughput: Figure 9 shows the throughput results in number of characters per second for different document sizes. Similarly to the latency results, when comparing the throughput between the SW case and the PCIe-FPGA version, we observe a minor improvement of 30%. In contrast, for network-attached FPGAs, the throughput increases by a factor of 14 for the two smaller document sizes and by a factor of 18 for the 2 kB case.

3) Latency Variation: We evaluated the latency variation by considering the latency of one million iterations for each document size. The standard deviation of the latency distribution is shown in Figure 10. The standard deviation of the latency ranges between 2.9 and 4.1 in the case of SW-only and PCIe-attached FPGA implementations, where as for the network-attached FPGAs it ranges between 0.5 and 0.8. Our preliminary experiments for inter-FPGA communication with raw TCP data showed that the standard deviation varies between 0.2 to 0.4, but in these experiments the master server, which generates and receive data contributes to the increment of the variation.

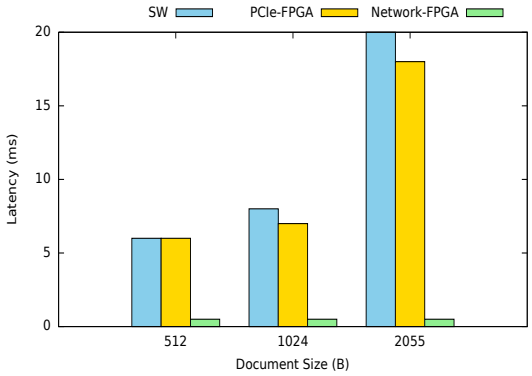

Fig. 8. Latency

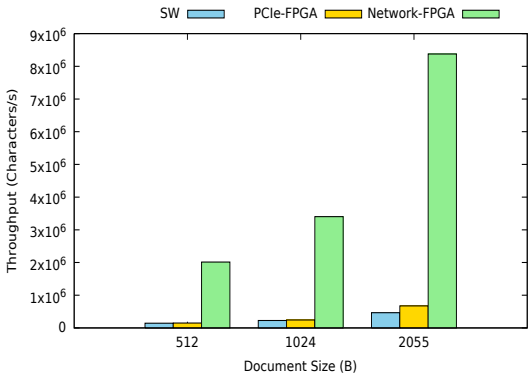

Fig. 9. Throughput

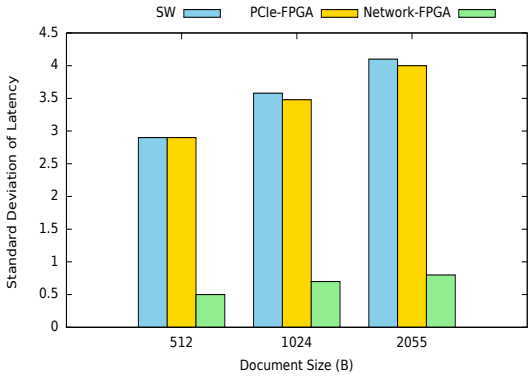

Fig. 10. Variation of response time

C. Discussion

1) Performance: The minor performance improvement observed when moving from SW only to PCIe-FPGA reflects the overhead of the scaleout framework. Because the actual processing time for running the regular expression in the FPGA is much shorter than in SW. The regular expression used for AEs requires 700μs on average in SW, while only taking 18μs in the FPGA. This is an 38 fold improvement, but the communication overhead of the framework still dominates the end result of latency, throughput, and latency variation. Compared with the SW-only version and the acceleration of the application with PCIe-attached FPGAs, the tight coupling of network packet processing and computation within the FPGA, and the elimination of the overhead of the framework when sending and receiving CAS objects (CAS-object creation, the serialization and de-serialization) lead to superior latency, throughput, and latency variation results for network-attached FPGAs.

An alternative way to setup the PCIe-attached FPGA experimental case is to add a sending AE and a receiving AE to the master server, similarly in the case of network-attached FPGAs. Even if we do this, the servers that host the PCIe-attached FPGAs have to execute the network packet processing, which significantly degrades the overall application

performance, particularly when the processing pipeline gets longer. Furthermore, a DC-class server consumes around 200 W of power, where as an FPGA device consumes around 25 W. Hence, moving from PCIe-attached FPGA implementation to standalone network-attached FPGAs, we achieve an order of magnitude improvement in power consumption.

TABLE I
RESOURCE CONSUMPTION

Module	LUT	FF	BRAM
NSL	88213	101122	429
AIL	1913	1768	15
MGL	621	458	9
vFPGA APP Wrapper	2761	1650	14
APP(RegEx)	5548	6807	2
Total	99056	111805	469
% of XC7VX690T	23	13	32

2) Resource Consumption: Table I shows the resource usage of the network-attached FPGA prototype with the application. The design uses around 99K LUTs, 111K Flip-Flops, and 469 BRAMS, which is equal to 23%, 13%, and 32% of the overall available resources, respectively. Out of the total resources, the NSL consumes around 20% of the LUT, 11% of the FF and 29% of BRAM resources, contributing the most for the resource consumption. In this work, the application we use consumes only a very low amount of resources, but in our future work we want to experiment with applications that fully use the FPGA resources.

VIII. CONCLUSION

The performance requirements of modern big-data applications are exceeding the performance offered by general-purpose servers in current data centers (DC). We believe that FPGAs have the potential to bridge this ever-increasing gap. We proposed a scalable and flexible system architecture to improve the processing of such applications using network-attached FPGAs. We implemented a prototype of a network-attached FPGA and formed a multi-FPGA fabric by connecting multiple network-attached FPGAs together over the DC network. We ported a text-analytics application, which runs on the UIMA distributed computing framework, onto this multi-FPGA fabric. We compared the results of our approach with a SW-only implementation and an implementation accelerated with PCIe-attached FPGAs. The comparison shows that our network-attached FPGAs outperform both these implementations in large margins, and improve the latency, the throughput, and the latency variation by a factor of 40, 18, and 5 respectively. The insights gained from these results open the way for running large-scale applications on network-attached FPGAs in DCs.

REFERENCES

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24.

[2] M. Blott *et al.*, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[3] J. Lockwood *et al.*, "A low-latency library in FPGA hardware for high-frequency trading (HFT)," in *2012 IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug 2012, pp. 9–16.

[4] J. Weerasinghe *et al.*, "Enabling FPGAs in hyperscale data centers," in *2015 IEEE International Conference on Big Data and Cloud Computing (CBDCom)*, August 2015, pp. 1078–1086.

[5] , "Apache hadoop." [Online]. Available: http://hadoop.apache.org/

[6] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[7] M. Isard *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007, pp. 59–72.

[8] D. Ferrucci and A. Lally, "UIMA: an architectural approach to unstructured information processing in the corporate research environment," *Natural Language Engineering*, vol. 10, no. 3-4, pp. 327–348, 2004.

[9] H. Giefers *et al.*, "Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid cpu/fpga system," in *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, June 2014, pp. 92–99.

[10] J. Dean *et al.*, "Large scale distributed deep networks," in *Neural Information Processing Systems, NIPS 2012*.

[11] M. Ebbers *et al.*, "Implementing IBM InfoSphere BigInsights on IBM System X," pp. 105–107. [Online]. Available: http://www.redbooks.ibm.com/

[12] Y. Guo *et al.*, "iShuffle: Improving Hadoop performance with shuffle-on-write," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, 2013, pp. 107–117.

[13] G. Ananthanarayanan *et al.*, "Effective straggler mitigation: Attack of the clones," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, Berkeley, CA, USA, 2013, pp. 185–198.

[14] S. Byma *et al.*, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '14, 2014, pp. 109–116.

[15] C. Chang *et al.*, "BEE2: a high-end reconfigurable computing system," *Design Test of Computers, IEEE*, vol. 22, no. 2, pp. 114–125, March 2005.

[16] O. Mencer *et al.*, "Cube: A 512-FPGA cluster," in *Programmable Logic, 2009. SPL. 5th Southern Conference on*, April 2009, pp. 51–57.

[17] T. Gneysu *et al.*, "Cryptanalysis with copacobana," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 57, no. 11, pp. 1498–1513, 2008.

[18] R. Baxter *et al.*, "Maxwell - a 64 FPGA supercomputer," in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, Aug 2007, pp. 287–294.

[19] Maxeler, "Maxeler MPC-X series." [Online]. Available: https://www.maxeler.com/products/mpc-xseries/

[20] L. Chiticariu, Y. Li, and F. R. Reiss, "Rule-based information extraction is dead! long live rule-based information extraction systems!" in *EMNLP*, 2013, pp. 827–832.

[21] R. Polig, K. Atasu, and C. Hagleitner, "Token-based dictionary pattern matching for text analytics," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–6.

[22] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss, "Hardware-accelerated regular expression matching for high-throughput text analytics," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–7.

[23] E. A. Epstein, M. I. Schor, B. Iyer, A. Lally, E. W. Brown, and J. Cwiklik, "Making watson fast," *IBM Journal of Research and Development*, vol. 56, no. 3.4, pp. 15–1, 2012.

[24] R. Polig, K. Atasu, H. Giefers, and L. Chiticariu, "Compiling text analytics queries to FPGAs," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014, pp. 1–6.

[25] Cyclone, "Pcie2-2711 - PCIe Gen2 eight slot expansion system." [Online]. Available: http://cyclone.com/pdf/600_2711%20datasheet.pdf