When FPGA-Accelerator Meets Stream Data Processing in the Edge

Song Wu, Die Hu, Shadi Ibrahim, Hai Jin, Jiang Xiao, Fei Chen, Haikun Liu

When FPGA-accelerator meets Stream Data Processing in the Edge

Song Wu[1], Die Hu[1], Shadi Ibrahim[2], Hai Jin[1], Jiang Xiao[1], Fei Chen[1], and Haikun Liu[1]

[1]*National Engineering Research Center for Big Data Technology and System*
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[2]*Inria, IMT Atlantique, LS2N, Nantes, France,*
Email: {wusong, hudie2016, hjin, jiangxiao, fei_chen_2013, hkliu}@hust.edu.cn, shadi.ibrahim@inria.fr

***Abstract*—Today, stream data applications represent the killer applications for Edge computing: placing computation close to the data source facilitates real-time analysis. Previous efforts have focused on introducing light-weight *distributed stream processing* (DSP) systems and dividing the computation between Edge servers and the clouds. Unfortunately, given the limited computation power of Edge servers, current efforts may fail in practice to achieve the desired latency of stream data applications. In this vision paper, we argue that by introducing FPGAs in Edge servers and integrating them into DSP systems, we might be able to realize stream data processing in Edge infrastructures. We demonstrate that through the design, implementation, and evaluation of F-Storm, an FPGA-accelerated and general-purpose distributed stream processing system on Edge servers. F-Storm integrates PCIe-based FPGAs into Edge-based stream processing systems and provides accelerators as a service for stream data applications. We evaluate F-Storm using different representative stream data applications. Our experiments show that compared to Storm, F-Storm reduces the latency by 36% and 75% for matrix multiplication and grep application. It also obtains 1.4x and 2.1x improvement for these two applications, respectively. We expect this work to accelerate progress in this domain.**

I. INTRODUCTION

In the era of *Internet of Everything* (IoE), there are billions of sensors and devices, usually at the Edge of the network, that are continuously generating high volume of stream data. It is predicted that there will be 50 billions interconnected *Internet of Thing* (IoT) devices which are expected to generate 400 Zetta Bytes of data per year by 2020 [1]. This led to the proliferation of *Distributed Stream Processing* (DSP) systems such as Storm [2], Flink [3], and Spark Streaming [4] in data centers and clouds to perform online processing of these continuous and unbounded data streams [5]. Despite that clouds can provide abundant computing resources to meet the ever-growing computation demands of streaming data applications, processing stream data in the clouds may come with high latency – this strongly depends on the transmission time from the data source to the clouds.

More recently, Edge infrastructure is gradually replacing clouds for low latency stream data processing such as video stream analysis [6, 7] and smart city applications [8]. The main reason of such transition is that moving computation to data sources can eliminate the latency of data transmission through *wide area network* (WAN) and therefore meet the latency requirements of many stream data applications. For instance, running smart city applications at the Edge shows to be up to 56% more efficient compared to clouds [9]. Accordingly, many efforts have focused on introducing *light-weight* DSP systems (i.e., F-MStorm [10], Apache Edgent [11], and Apache MiNiFi[12]), thus, Edge resources concentrate on data processing (DSP platforms introduce low overhead) and service provisioning can be performed fast. On the other hand, given the limited computation power of Edge servers, some work is dedicated to enable a unified run-time across devices, Edge servers and clouds, thus stream data processing can be performed in both Edge servers and clouds [13, 14]. Unfortunately, this inherits the same high latency introduced by clouds. *Hence, current efforts may fail in practice to achieve the desired latency of stream data applications.*

There are two ways to mitigate the limit in computation power of current Edge servers: (i) scaling-out and (ii) scaling-up Edge servers. Scaling-out may result in high latency: the latency is dominated by the performance of inter-operator communications, which is in turn limited by the network bandwidth among Edge servers [15]. Scaling-up by simply adding *general-purpose processors* (CPUs) is hard and comes with several challenges including large and complex control units, memory wall and Von Neumann bottleneck, redundant memory accesses, and power consumption [16]. Efforts on adopting accelerators acknowledge those challenges [16, 17] and aim to exploit their powerful parallel computing capabilities efficiently.

In this work, we focus on *Field Programmable Gate Arrays* (FPGAs) because FPGA is an ideal candidate for steam data processing in the Edge: First, unlike *Graphic Processing Unit* (GPU) which only provides data parallelism, FPGAs can provide data, task, and pipeline parallelism and therefore are more suitable for stream processing and can serve a wider range of IoT applications [1]. Second, FPGA promises high computation power and at the same time sustains much lower energy consumption compared to GPU [18] which is a critical requirement for Edge servers. Third, FPGAs are widely deployed accelerators in data centers (e.g., Microsoft have deployed FPGAs on 1632 servers to accelerate the Bing web search engine [19]) and have been successfully used to accelerate data mining [20], graph processing [21], deep learning [22, 23], and stream data processing [24–26].

In an attempt to demonstrate the potential benefits of exploiting FPGAs for stream data processing in the Edge, in this vision paper, we present the design, implementation,

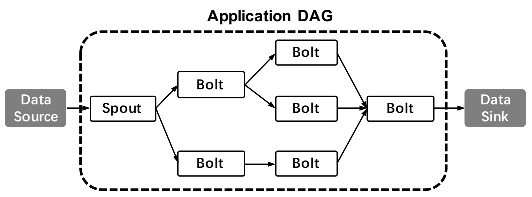


Fig. 1. A stream data application DAG

and evaluation of F-Storm, an FPGA-accelerated and general-purpose distributed stream processing system in the Edge. By analyzing current efforts to enable stream data processing in the Edge and to exploit FPGAs for data-intensive applications, we derive the key design aspects of F-Storm. Specifically, F-Storm is designed to: (1) provide a *light-weight* integration of FPGA with a DSP system in Edge servers, (2) make full use of FPGA resources when assigning tasks, (3) relieve the high overhead when transferring data between *Java Virtual Machine* (JVM) and FPGAs, and importantly (4) provide programming interface for users that enable them to leverage FPGA accelerators easily while developing their stream data applications. We have implemented F-Storm based on Storm. Evaluation results show that F-Storm reduces the latency by 36% and 75% for matrix multiplication and grep application compared to Storm. Furthermore, F-Storm obtains 1.4x, 2.1x, and 3.4x throughput improvement for matrix multiplication, grep application, and vector addition, respectively. In addition, F-Storm reduces the CPU utilization of the main threads of offloaded operators in stream data applications significantly. We expect this work to offer useful insight into leveraging FPGAs for stream data application in the Edge and to accelerate progress in this domain.

The rest of the paper is organized as follows. In Section 2, we introduce the background including stream data processing, Edge computing, and FPGAs. Then we analyze how to unleash the power of Edge for stream data processing in Section 3 and discuss the design goals of F-Storm in Section 4. Next, we describe the system design and implementation in Section 5 and present our experimental results with corresponding analysis in Section 6. Section 7 discusses lessons learned and future directions. Finally, we conclude the paper in Section 8.

II. BACKGROUND

A. Stream Data Processing

Over the years, stream data processing has attracted much attention with the explosion of big data and the increase of real-time requirements. In this subsection, we will briefly introduce stream data applications and present the state-of-the-art DSP systems.

Stream data application. Stream data application is usually abstracted as a *Directed Acyclic Graph* (DAG) in which each vertex represents an *operator* with user-defined computation logic and each *edge* represents a *data stream*. Figure 1 shows a typical stream data application (called Topology in Storm) which contains two types of operators: *spout* and *bolt*. A spout reads data from external data sources and then emits them into the topology. A bolt consumes data from spouts or other bolts, processes them, and then emits new data streams or finally outputs its results into data sinks. Each spout or bolt can be instantiated into multiple tasks which can be executed on several executors in parallel.

DSP systems. Various DSP systems such as Apache Storm [2], Flink [3], and Spark Streaming [4] have been proposed to develop and execute stream data applications. They are usually built on a cluster or cloud, combining resources of multiple physical servers to process data streams continuously. Each of them has different runtime architecture. The common runtime architecture is a *master-worker* architecture which consists of a master node and multiple work nodes. The master node is the central node of the architecture which is responsible for controlling the overall runtime environment, the reception of user-submitted jobs, and scheduling tasks. Each worker node receives and performs the tasks assigned to it by the master node.

B. Edge Computing

Edge computing refers to a new technology that enables computation to be performed near the data sources. Edge not only requests services and contents from clouds, but also performs computing tasks. Edge includes any computing resources which are placed near the data sources, such as a cloudlet [27] or a micro data center. Compared to cloud, Edge usually has less compute and storage resources. However, if data can be processed at the Edge, this avoids expensive data transmission and greatly reduces network pressure. Importantly, Edge computing can help to eliminate the high network latency when transferring data from the data sources to clouds, and can serve a wide variety of time-sensitive IoT applications. For example, the response time of face recognition application is reduced from 900ms to 169ms by offloading computation from cloud to the Edge [28]. Moreover, Edge computing can save much energy because there is no need to always upload data to cloud.

C. FPGAs: Field Programmable Gate Arrays

FPGA consists of millions of logic blocks and thousands of memory blocks which are interconnected by a routing fabric [29]. The logic blocks on FPGA are equivalent to thousands of “CPU cores” that can perform parallel computations and the memory blocks can provide high memory bandwidth. Therefore, FPGAs can provide massive parallelism that can be used to accelerate large-scale parallel computing applications. Recently, FPGAs have been widely used in accelerating graph processing [30], machine learning [31], and so on. Furthermore, FPGAs have been deployed in many large data centers like Microsoft [19] to accelerate high performance computing. There are two major vendors of FPGAs: Xilinx and Altera (later took over by Intel). They have released their respective OpenCL-based [32] high-level programming tools (Xilinx SDAccel Development Environment and Altera SDK for OpenCL [33]) which can lower the hardware knowledge threshold of FPGA development and improve the development efficiency.

III. Unleash the Power of Edge for Stream Data Processing

This section concisely reviews the main research efforts on enabling stream data processing in the Edge and the main systems which leverage FPGAs for big data applications. This serves to derive the main requirements toward achieving effective stream data processing in the Edge.

A. Stream Data Processing in Edge: Current State and Desired Features

Given the practical importance of enabling stream data processing in the Edge, several DSP systems are proposed to enable mobile devices to perform stream data processing including MStorm [34], F-MStorm [10], Symbiosis [35], and Mobistreams [36]. Apache Edgent [11] and Apache MiNiFi [12] are introduced to enable performing stream data processing on low performance IoT devices. Seagull [37] is built towards resource-constrained Edge servers and uses a location and capacity-aware strategy to distribute stream data processing tasks to multiple Edge servers. DART [38] is a light-weight stream processing framework for IoT environment which tries to execute tasks on the Edge devices and Edge servers. *An important goal of the aforementioned DSP systems is to reduce the overhead of current deployment of stream data engines (e.g., Spark Streaming requires several services to be running including Spark and Hadoop), therefore Edge resources concentrate on data processing, and service provisioning can be performed fast.* Hence, **(REQ1) DSP systems for Edge servers should be *light-weight* and featured with low running and deployment overhead.**

Previous DSP systems enable performing stream data processing on Edge servers but can not handle the whole data stream [13, 39–43]. It is the job of the system administrator to decide the size of the stream which can run on Edge servers and to coordinate the results. To reduce the burden of manually dividing the streams and coordinating the execution between Edge servers and clouds, many research efforts have been dedicated to seamlessly execute (and optimize the execution of) stream data applications on both Edge severs and clouds. Amino [39] provides a unified run-time across devices, Edge, and clouds. SpanEdge [13] is a novel framework that provides a programming environment to enable programmers to specify the parts of their applications that need to be close to the data sources in order to reduce the latency and network cost. Moreover, new placement strategies are introduced to schedule operators across Edge servers and clouds to reduce the response time and the cost of data transmission through network [42, 43]. *While these efforts acknowledge the limitation of Edge servers to handle stream data applications (due to the limited computation power on Edge servers), they approach the problem by scaling-out the computation to clouds. Although indeed important, they provide a little improvement compared to transmitting the whole stream to the cloud – this strongly depends on the number of tasks which the Edge servers can accommodate and the cost of the inter-operation communication [15].* Hence, **(REQ2) Scaling-out comes at a high cost of inter-operation communication (when operators located on the Edge and clouds exchange data) and is limited to the computation which Edge servers can afford, thus if we would like to effectively process stream data in Edge servers, we should investigate scaling-up approaches [44], specifically increasing the computation power of Edge servers.**

B. On the Role of FPGAs for Stream Data Processing in the Edge

FPGA presents an ideal candidate for steam data processing in the Edge as it provides data, task, and pipeline parallelism and thus can serve a wider range of IoT applications [1]. Moreover, FPGA features with low energy consumption which is a critical requirement for Edge servers. Hereafter, we summarize current efforts to leverage FPGAs for data-intensive applications in an attempt to derive the main missing puzzle pieces for FPGA-based DSP systems in the Edge.

Leveraging FPGAs for MapReduce applications has been introduced in [45–47]. Despite that those systems are limited to map and reduce functions, they reveal an important challenge on the complexity of writing customized programs to expose the hardware details on FPGAs. Given the wide adoption of Spark as the *de-facto* engine for iterative applications, several efforts have investigated the benefits of applying FPGA accelerators to Spark [26, 48, 49]. For example, Chen et al. [48] show how applying FPGA accelerators to Spark running on CPU-cluster can achieve 2.6x performance improvement for DNA sequencing application. Blaze [26] integrates FPGAs into Spark and Yarn and thus can provide programming and runtime support that enables users to leverage FPGA accelerators easily, but at the cost of high overhead due to extra layers to manage the FPGA resources. *While performance gain is undebatable when using FPGAs, exposing FPGAs is not an easy task.* Hence, **(piece 1) An FPGA-based DSP system for Edge should hide the hardware details of FPGA from applications' developers without adding extra heavy-weight layers which may increase the overhead of the system.**

Few efforts have focused on integrating FPGAs into DSP systems to accelerate stream data applications. In [24], Maxeler MPC-C Series platform – special FPGA platform – is integrated within Apache Storm. Another work [25] combines Spark Streaming and FPGA NIC. A recent project called HaaS [50] presents a novel stream processing framework which integrates CPUs, GPUs and FPGAs as computation resources. HaaS uses general PCIe-based FPGAs and focuses on heterogeneity-aware task scheduling to expose FPGA and GPU resources efficiently. **(Piece 2) To allow large deployment in the Edge, DSP systems should be integrated into general PCIe-based FPGAs and consider resource heterogeneity when scheduling tasks.** Finally, as most current stream data processing are built on JVM, **(Piece 3) it is important to consider and reduce JVM-FPGA communications.**

IV. F-Storm: FPGA Meets Stream Data Processing in the Edge

This paper aims to show the practical importance of utilizing FPGAs in Edge servers to realize stream data processing close to data sources, and to draw insights to facilitate future research. Accordingly, we design F-Storm, an FPGA-accelerated and general-purpose distributed stream processing system on the Edge. Hereafter, we summarize the design goals of F-Storm which carefully answer the requirements and issues discussed in Section 3.

Light-weight FPGA integration. The gap between the JVM-based DSP systems and the C/C++/OpenCL-based FPGA is the first problem which must be solved. Unlike previous works which require extra layers (libraries) to expose FPGA resources [24–26, 49, 50], F-Storm incorporates a light-weight way to integrate PCIe-based FPGAs into DSP systems: F-Storm is a standalone FPGA-accelerated DSP system which can be deployed with low cost. Hence, Edge servers can concentrate their computation power on processing stream data.

Accelerator-prioritized scheduling. To utilize FPGAs in a distributed and heterogeneous environment, DSP systems should be aware of FPGA resources and accordingly identify and assign tasks to FPGA accelerators. However, most existing schedulers of DSP systems consider only CPUs as the computing resources or provide coarse-grained scheduling [50]. F-Storm is equipped with a fine-grained scheduler which makes full use of FPGA resources when scheduling tasks and can efficiently continue the execution of the applications even when FPGA resources are not sufficient which is important in Edge environment.

DSP-targeted JVM-FPGA communication mechanism. Previous works on adopting FPGAs in DSP systems focus on how to accelerate the computation but overlook the cost of transferring data between JVMs and FPGAs [24, 50]. F-Storm is equipped with a DSP-targeted JVM-FPGA communication mechanism which adopts two well-known techniques including data batching and data transfer pipelining to reduce the overhead of JVM-FPGA data transmission, and therefore allows to expose FPGAs in Edge servers effectively and meet the latency demands of different IoT stream data applications.

User-friendly programming interface. In order to make it easy for users to leverage FPGA accelerators in stream data applications, F-Storm provides a simple-to-use programming interface for users to develop their applications and exploit FPGA accelerators easily. The programming interface hides many complicated details about FPGA accelerators so that users can focus on the implementation of the accelerator kernel computing logic and the setting of relevant parameters. This will not only result in a great reduction of the lines of codes when developing an application to utilize FPGAs but also allow large range of (IoT) stream data applications to run in the Edge.

V. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of F-Storm which is developed based on Storm. We start with an overview of F-Storm and then go into system details.

A. System Overview

F-Storm architecture. F-Storm is designed to be deployed in an Edge cluster which contains only a small number of Edge servers. As shown in Figure 2, several Edge servers form a small F-Storm cluster. The runtime architecture of F-Storm is a typical *master-worker* architecture in which one Edge server acts as the master node and other Edge servers equipped with several FPGA cards play the role of worker nodes. The master node manages all the resources in the cluster including FPGAs and uses an *Accelerator-prioritized Scheduler* to perform task scheduling. Each worker node runs a JVM daemon called *supervisor* which is responsible not only for listening to the job assignments and managing worker processes, but also monitoring the FPGA resources of the local node and reporting the related information to the master node. Because JVM can not access FPGA accelerators directly, we design a unified FPGA manager called *NodeFPGAManager* running on the native machine and add a new component called *FPGAInfoCommunicator* for each supervisor. Given the resource-limitation of Edge servers, NodeFPGAManager is designed to be light-weight, its deployment and running do not add large overhead to the Edge servers. NodeFPGAManager is responsible for managing FPGA resources, it can query and manipulate FPGAs through the APIs which are provided by Altera SDK for OpenCL [33]. FPGAInfoCommunicator can obtain FPGA accelerators' information indirectly by communicating with NodeFPGAManager and provide awareness of FPGA resources to the supervisor which it is attached to.

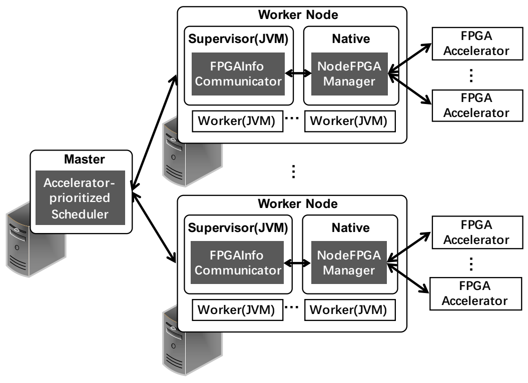


Fig. 2. High-level architecture of F-Storm

Workflow. A stream data application in F-Storm is abstracted as a topology which contains three types of operators: *spout*, *bolt*, and *accBolt*. While the spout and bolt are general components (we have introduced in Section 2), the accBolt represents a new type of operator which utilizes FPGA resources to process tasks. Stream operators which utilize FPGAs are implemented as accBolts. Before users submit an F-Storm topology containing one or several accBolts, they need to write an additional kernel function that specifies the processing logic for each accBolt. All the kernel functions of a topology are put into an OpenCL file and then users need to compile the file into a file with the suffix *.aocx* which can be executed on FPGA cards. We summarize the overall workflow of F-Storm as follows.

1. The user submits a topology that contains some general operators and one or multiple accBolt operators, each with a kernel function written in OpenCL.
2. The master node distinguishes and tags the two types

of operators and then uses the accelerator-prioritized scheduler to assign tasks to worker nodes.

3. Each worker node receives job assignments and launches worker processes to run one or multiple executors which are JVM-threads. The executors of accBolt operators are called *acc-executor*, they are responsible for offloading the computation to FPGA accelerators while others are called *general-executor* which are used to run tasks of general operators on CPUs directly.
4. When data needs to be moved from a general-executor to an acc-executor, the acc-executor buffers the input tuples into batches and transfers them to the FPGA to be executed. After getting the signal that the data outputs are ready, the acc-executor pulls the results, reconstructs them into tuples, and emits them to the downstream tasks.

B. Light-weight FPGA Integration

We design *NodeFPGAManager*, a light-weight and pluggable module located on each worker node. It can manipulate FPGA resources and provide the accesses of FPGA accelerators to a number of threads of multiple stream data applications. We can turn on and off NodeFPGAManager by setting the parameters in the configuration file. In this way, F-Storm can easily manage all the resources of the cluster by itself, independent of other heavy-weight resource management frameworks such as Yarn [51] and Mesos [52]. Compared to Blaze [26] whose deployment requires Spark, Yarn, Hadoop, Intel AALSDK, Boost, and Google Protobuf, deploying F-Storm requires only Zookeeper and Altera SDK for OpenCL (Intel SDK for OpenCL), which leads to lower deployment cost.

Specifically, NodeFPGAManager plays two roles. One acts as an FPGA-related information reporter for the JVM runtime of F-Storm and another performs as an FPGA manipulator for FPGAs. As an information reporter, NodeFPGAManager reports information about all the FPGA accelerators to the JVM runtime of F-Storm. Each NodeFPGAManager can get all the FPGA resource information of the local node at startup and report them to the worker node through communicating with FPGAInfoCommunicator. Each worker node will be aware of the FPGA resources of the local node and can send these information to the master node through heartbeat mechanism.

As an FPGA manipulator, it responds to FPGA accelerator requests from the threads of stream data applications, providing access to the FPGA accelerators for them. The workflow of handling the requests of FPGA accelerator can be summarized as three steps: First, NodeFPGAManager analyzes the request and extracts various parameters related to the FPGA accelerator to be executed. Second, NodeFPGAManager assigns an FPGA accelerator to the request and starts a host program to control the FPGA accelerator including kernel startup and data moving. Meanwhile, the state of the assigned FPGA accelerator is changed from idle to busy. Third, when NodeFPGAManager receives a signal to stop an accelerator, it shuts down the kernel task running on the FPGA and stops itself safely. Finally, the FPGA accelerator returns to idleness to inform that it can be used again.

Algorithm 1 Accelerator-prioritized scheduling algorithm.

```
Input:
    the set of general-executors, $Eg$,
    the set of acc-executors, $Ea$,
    the set $F = F_k$, $F_k$ represents the number of available FPGA
    devices on the $k$th worker node,
    the set $S = S_k$, $S_k$ represents the number of available slots on
    the $k$th worker node
Output:
    the set X = (ai, sj), a pair (ai, sj) means an acc-executor i is
    assigned to slot sj,
    the set Y = (gi, sj), a pair (gi, sj) means a general-executor i is
    assigned to slot sj,
 1: for $F_i$ in $F$ do
 2:     if $S_i == 0$ then
 3:         $F_i \leftarrow 0$;
 4:     end if
 5: end for
 6: $ft \leftarrow \sum_{i=1}^{Ns} F_i$;
 7: if $ft < |Ea|$ then
 8:     for $i = ft + 1$ to $|Ea|$ do
 9:         convert $Ea_i$ to a general-executor and put it into $Eg$;
10:     end for
11: end if
12: for $Ea_i$ in $Ea$ do
13:     $s \leftarrow argmax(F_k)$;
14:     $w \leftarrow$ available slot in the worker node $s$;
15:     add$(Ea_i, w)$ to X;
16:     Update $F_k$ with $F_k - 1$;
17: end for
18: for $Eg_i$ in $Eg$ do
19:     $s \leftarrow argmax(S_k)$;
20:     $w \leftarrow$ available slot in the worker node $s$;
21:     add$(Eg_i, w)$ to Y;
22:     Update $S_k$ with $S_k - 1$;
23: end for
```

C. Accelerator-prioritized Scheduling

Edge servers are limited in their computation capacities. The FPGA cards which are installed on the Edge server are also limited, thus there may be insufficient FPGA resources when running a stream data application. The state of resources, including FPGA accelerators, are hidden to users, so users may develop and submit some applications that request more FPGA resources than the total amount of available devices in the Edge cluster, which may cause failures. To prevent failed execution of such applications and make full use of FPGA resources, we design a flexible accelerator-prioritized scheduler.

When performing task scheduling, all the operators in a stream data application are converted to one or multiple executors which execute the processing logic of them. First, the scheduler divides these executors into two sets. The executors of all the accBolts are labeled as ***acc-executors*** which request to use FPGA accelerators and the executors of other general operators are labeled as ***general-executors*** which only use CPUs. Then, the scheduler gets the total number of available slots (Each slot can run a worker process) and the total number of available FPGA devices in the cluster. Next, the scheduler assigns the two types of executors to multiple slots. The accelerator-prioritized scheduling strategy attempts to assign as many acc-executors as possible to the worker nodes that have available FPGA devices and slots, maximizing the utilization of FPGA resources. Then the remaining acc-executors can be switched to general-executors and be assigned to all available worker nodes together with other general-executors.

The detailed scheduling process is listed as Algorithm 1. The number of worker nodes is labled as ***Ns***. If there are no available slots on a worker node, all the FPGA devices attached to it are also unavailable. Hence the first few lines (1 to 5) of code are to set the number of available FPGAs to 0 for those worker nodes with no slots to use, and then we can get the actual total number of available FPGA devices of the cluster as shown in line 6. Next from line 7 to 10, we convert several acc-executors to general-executors if the FPGA devices are not sufficient. After that, all executors are finally divided into two sets: acc-executors and general-executors. Lines from 12 to 17 and lines from 18 to 23 are to assign acc-executors and general executors to all the worker nodes, respectively. This scheduling strategy makes full use of FPGA resources when allocating tasks. Furthermore, it takes load balancing into consideration while assigning tasks.

D. DSP-targeted JVM-FPGA Communication Mechanism

FPGAs can help to improve the performance of stream data applications by accelerating the computation of tasks, but, the performance gain can be offset by the overhead of JVM-FPGA data moving. Therefore, we propose a DSP-targeted JVM-FPGA communication mechanism which adopts two well-known techniques including data batching and data transfer pipelining to mitigate the extra overhead.

Data batching. In F-Storm, each task processes incoming tuples one by one. Transferring small-size tuples to FPGA one by one can result in extremely low communication bandwidth that may cause significant performance loss [48]. So it is necessary to buffer a group of small-size tuples together and transfer them to FPGAs simultaneously. Then the FPGA accelerator processes data batch by batch and passes the results back to JVM threads. The batch size is a user-configurable parameter. In particular, if users set the batch size to 1, it means that once a tuple comes, it will be transferred to FPGA immediately to be processed instead of being put into the input buffer. When the data size of a single tuple is small, it is preferred to set the batch size to a large value, and when a single tuple has a large amount of data, users should configure a small value for the batch size.

Data transfer pipelining. The sequential execution of the complex JVM-FPGA data communication routine usually results in large overhead (long response time) [53]. F-Storm adopts data transfer pipelining to alleviate the large overhead. In short, the technique aims to form a multistage data transfer pipelining that contains several data transmission steps, which can be seen in Figure 3. In this procedure, the JVM process accepts multiple tuples which are Java objects and transfers them through three pipelining steps (① ② ③) to the FPGA accelerators. After the computation is completed, the data outputs are transferred back to a JVM, also through three pipelining steps (④ ⑤ ⑥). Hereafter, we describe in details the first three pipelining steps which transfer data from JVM to FPGA. Note that, the remaining steps are processed in similar way.

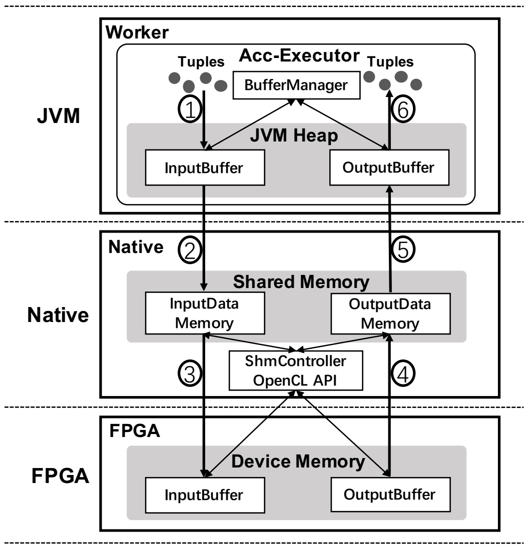


Fig. 3. The overview of JVM-Native-FPGA data transmission

- **Extract and cache (①).** The objective of this stage is to extract and cache the input data of multiple tuples to improve the data transfer bandwidth. The acc-executors which run on CPU worker process execute tasks of accBolt operators whose actual computations need to be offloaded to FPGAs. While an acc-executor is initialized, input buffers and output buffers are allocated with a *BufferManager*. Once a tuple comes, the acc-executor extracts the data of the tuple and caches them in input buffers until the input buffer is full.
- **Send to native memory (②).** The step aims to send the input data from JVM to the native memory. Once the input data buffer is full, the BufferManager transfers the batched data from input buffers located on JVM heap to the native buffers which are controlled by native processes. This stage involves the intra-node *Inter-Process Communication* (IPC) between the JVM-based process and the native process. In our implementation, we choose to use shared memory to do JVM-Native data moving. This ensures fast intra-node IPC.
- **Send to FPGA (③).** This stage gathers the data in the native shared memory and transfers them to the memory of FPGA devices using Altera OpenCL APIs. During this stage, an important manager called *ShmController* takes charge of manipulating the shared memory. Once the input data is ready, ShmController triggers the data transmission from shared memory to FPGA devices. Then the execution on FPGA can be started.

E. User-friendly Programming Interface

In Edge environment, the diversity of stream data applications makes it important to provide a user-friendly programming interface for users to develop their applications quickly. F-Storm achieves this goal by providing easy-to-use *topology programming interface* and simplifying *FPGA accelerator* *programming*. With a few changes to the application code, users can leverage FPGA accelerators in their stream data applications.

Topology programming interface. Topology programming model enables users to leverage FPGA accelerators in their stream data applications with minimal code changes. To this end, we design and implement *BaseRichAccBolt* which supports offloading computation to FPGA accelerators automatically. When users want to accelerate an operator, they can construct an accBolt by inheriting the BaseRichAccBolt and then the accBolt can be instantiated as an acc-executor which can do computation offloading. BaseRichAccBolt implements the basic interface called *IAccBolt* which is demostrated in Listing 1. The interface contains three functions. The *accPrepare* method is called when initializing the acc-executor and it is responsible for notifying the BufferManager to allocate and initialize input and output buffers. The *accExecute* method uses the BufferManager to buffer multiple input tuples and transfer them in batches to the FPGA accelerator, and at the same time it creates another thread to wait for the batch results and emit them. Last, when the acc-executor needs to be shut down, the *accCleanup* method is called to clean up involved resources.

FPGA accelerator programming. The FPGA accelerator programming is to develop computation kernels using OpenCL [32]. Each accBolt in the topology corresponds to a kernel function in an OpenCL file. F-Storm hides the details of FPGA accelerator management including initialization and data transmission, so users only need to focus on implementing the computing logic of the kernel, which can save a lot of lines of code. A simple vector addition kernel is implemented in Listing 2. Users need to specify the parallelism of the computation while implementing the kernel function so that it can be executed on the FPGA efficiently.

Listing 1. IAccBolt.java

```
public interface IAccBolt extends
    Serializable{
        void accPrepare(Map stormConf,
             TopologyContext context,
             OutputCollector collector);
        void accExecute(Tuple input);
        void accCleanup();
}
```

Listing 2. Vector addition kernel

```
__kernel void vector_mult(
        __global const float *x,
        __global const float *y,
        __global float *restrict z)
{
        int index = get_global_id(0);
        z[index] = x[index] + y[index];
}
```

VI. EVALUATION

A. Experimental Setup

Given that F-Storm is designed for Edge clusters which have limited Edge servers, our experiments run on a small cluster of 3 physical servers, including a master node and two worker nodes. Each node has Intel Xeon E5-2630 v4 CPUs and 64GB main memory. Each worker node is equipped with an Intel Arria 10 FPGA via PCI-E slot. We implement F-Storm based on Storm 1.0.3, Altera SDK for OpenCL (version 16.0.0), and JNI technology. The compilation of the OpenCL file which contains one or multiple kernel functions relies on the Altera FPGA offline compiler which is provided by Altera SDK for OpenCL.

We use three representative applications: *Matrix Multiplication*, *Grep*, and *Vector Addition*. We evaluate three performance metrics. (1) Throughput: the number of tuples processed and acknowledged within a fixed time interval. (2) Latency: the average time that a tuple takes to be acknowledged. (3) CPU utilization of some key threads for these applications. We also measure lines of code of the applications to reflect the programming efforts. We compare F-Storm with the Storm 1.0.3 which is deployed on the same servers but doesn't utilize FPGA cards.

B. Matrix Multiplication

Matrix multiplication is a numerical computing application, which is used in graph processing [54]. We design a stream-version sequential topology consisting of three operators for it. The first one is a spout called *MatrixGeneratorSpout*, which constantly generates two matrices, puts them in a tuple, and emits it to downstream bolt. The second operator called *MatrixMultiplyBolt* is the actual bolt which extracts the two matrices and performs the multiplication operation. In Storm, it is implemented as a general bolt which runs on CPU while in F-Storm it is an accBolt which inherits BaseRichAccBolt and can request FPGA accelerator to do the computing work. The last operator is a bolt called *ResultWriterBolt*, which writes the results to a file.

The multiplication of two matrices is computation-intensive and F-Storm can leverage FPGA accelerators to compute each element or a group of elements of a result matrix in parallel. We implement an OpenCL kernel for MatrixMultiplyBolt which is executed on FPGA, computing a result matrix of two matrices' multiplication in parallel each time. In this case, the batch size is 1 which means once MatrixMultiplyBolt receives one tuple that contains two matrices, it sends the data to FPGA accelerator to be computed. The execution time of the kernel correlates with the input data size, which is directly decided by the matrix size in this application. Larger matrix size means larger input data size. Thus, we conduct sensitivity analysis of matrix size to find out the trend of performance changes with respect to the input matrix size.

First, on both Storm and F-Storm, we test the average tuple processing latency and the CPU utilization of the main thread in MatrixMultiplyBolt using different matrix sizes with a fixed input rate. As the results shown in Figure 4 and Figure 5, F-Storm achieves lower latency and lower CPU utilization compared to Storm. Moreover, the matrix size can impact the performance. When the matrix size is 128x128, the tuple processing latency of F-Storm is about 20ms while the one of Storm is 24ms (F-Storm reduces the latency by 16% compared to Storm). The CPU utilization of the key thread in F-Storm is only 2.9% while it is 11.4% in Storm, which means F-Storm

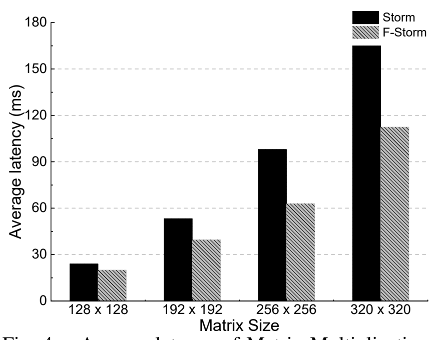


Fig. 4. Average latency of Matrix Multiplication

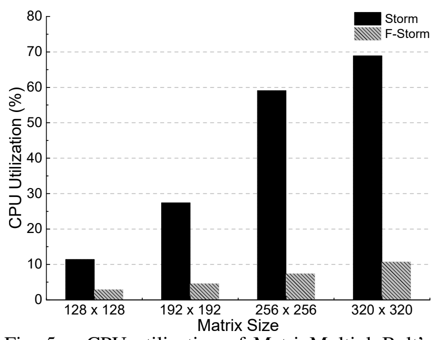


Fig. 5. CPU utilization of MatrixMultiplyBolt's main thread

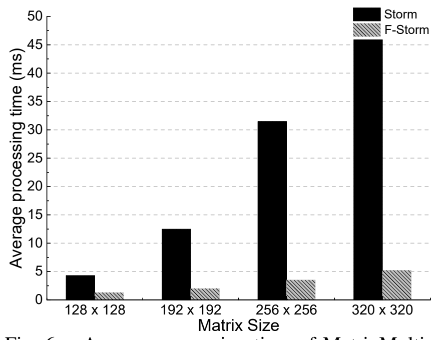


Fig. 6. Average processing time of MatrixMultiplyBolt

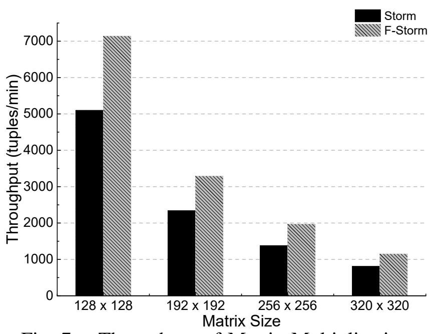


Fig. 7. Throughput of Matrix Multiplication

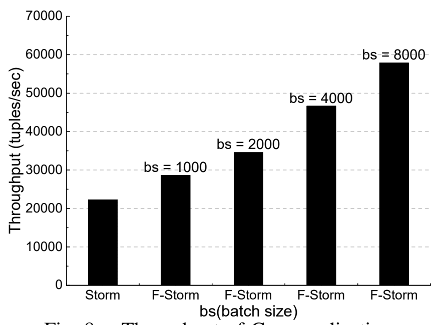


Fig. 8. Throughput of Grep application

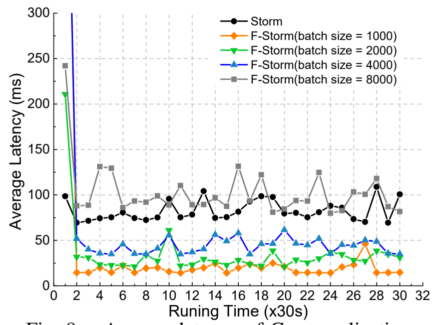


Fig. 9. Average latency of Grep application

uses less CPU time. Before the matrix size reaches 320x320, the performance improvement increases significantly with the increase of the matrix size, especially when the matrix size is 256x256, F-Storm reduces the average tuple processing time by 36% and lowers the CPU utilization from 51.1% to 7.4%. However, when the matrix size is set to 320x320, the reduction rate of tuple processing latency is decreased to 32%. This is due to the high cost of data transmission.

Next, we collect the average processing time of MatrixMultiplyBolt for Storm and F-Storm respectively and present the result in Figure 6. We can observe that compared to Storm, the average processing time of MatrixMultiplyBolt is much lower in F-Storm, and with the matrix size increasing, the decrease of the time is more obvious. This reflects that F-Storm achieves the kernel speedup and thus reduces the tuple processing latency. As for the decrease of CPU utilization, the reason is that F-Storm offloads the computation including vast multiplication and addition to FPGAs, so the main thread on CPU only needs to do data caching and transmission, which take less CPU time.

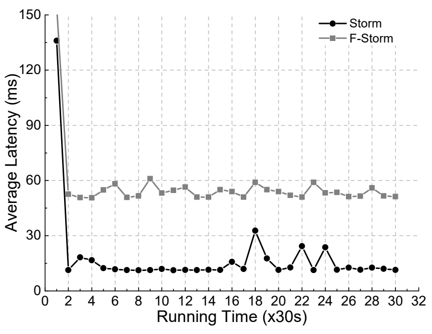
Last, we test the max throughput in different matrix sizes and show the results in Figure 7. We can find that compared to Storm, F-Storm achieves around 1.4x throughput improvement when the batch size is 256x256. *In summary, we observe that using F-Storm in Edge servers can speed up the entire application by accelerating computation-intensive operations. Moreover, this performance gain comes with low overhead: it can achieve much lower latency and lower CPU utilization while keeping higher throughput compared to Storm.*

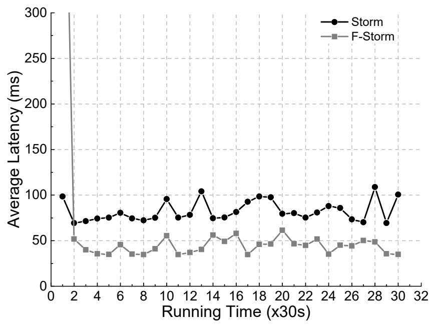
C. Grep

Grep is often used to evaluate the performance of DSP systems. In our topology, a spout called *SentenceGeneratorSpout* generates sentences randomly and emits them to downstream tasks. *SplitSentenceBolt* receives sentences and splits them into multiple words. Because SplitSentenceBolt converts one input tuple into multiple output tuples so it results in a heavy load for next bolt called *FindMatchingWordBolt*. FindMatchingWordBolt finds the matching words and emits them. In particular, it is implemented as a general bolt in Storm and an accBolt in F-Storm. The last bolt called *CountMatchingBolt* counts the matching words.

In this application, FindMatchingWordBolt implemented in F-Storm caches a group of tuples and transfers them to the FPGA to be processed, so the batch size is a key parameter which can have direct impact on the batch processing time and data transmission time. In the first experiment, we conduct sensitivity analysis of batch size to see the trend of performance changes with respect to the batch size. Given that FindMatchingWordBolt receives large amounts of small tuples and FPGA has large on-chip memory, we choose relatively large batch sizes to improve the data transmission bandwidth and make full use of parallel computing power of FPGA. The throughput result is shown in Figure 8. F-Storm achieves better throughput than Storm. When batch size is set to 4000, it obtains a high throughput of up to 46685 tuples/sec, around 2.1 times the throughput achieved by Storm. The reason for this result is that compared to the sequential processing of tuples by CPU, FPGA can compute a batch of tuples in parallel, thus reducing the tuple batch completion time and improving the throughput. However, when the batch size is increased to 8000, although the throughput is much higher, it results in higher latency under 20K tuples/sec input data rate as shown in Figure 9.

Figure 9 illustrates the tuple processing latency under different batch sizes with a fixed input data rate of 20K tuples/sec.


(a) Under 10K tuples/sec


(b) Under 20K tuples/sec

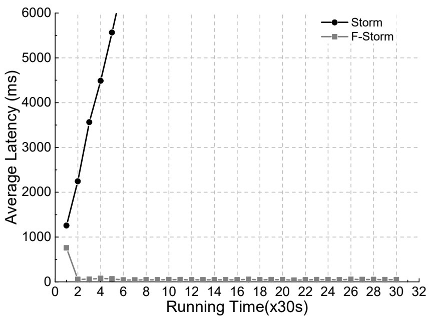

(c) Under 30K tuples/sec

Fig. 10. Average latency of Grep Application

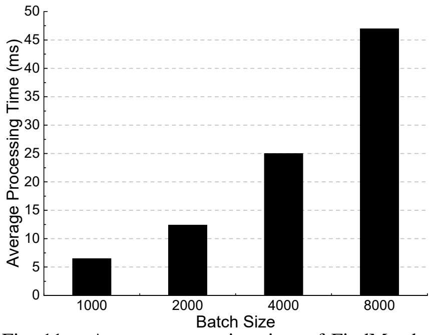


Fig. 11. Average processing time of FindMatchingWordBolt in F-Storm

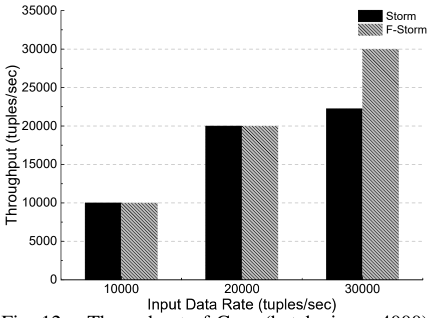


Fig. 12. Throughput of Grep (batch size = 4000)

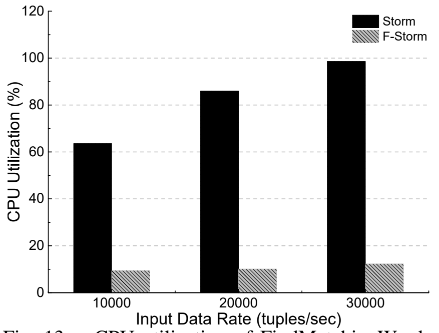


Fig. 13. CPU utilization of FindMatchingWordBolt's main thread

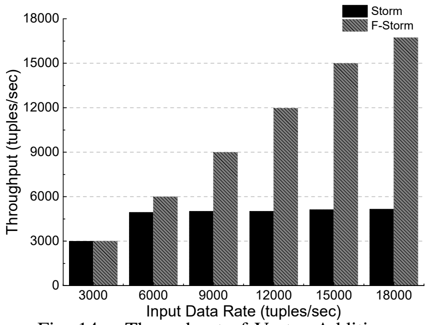


Fig. 14. Throughput of Vector Addition

We can see that both Storm and F-Storm show stable latency below 150 ms and the latency of F-Storm increases when increasing the batch size. When the batch size is set to a relatively small value, F-Storm has a lower latency compared to Storm, especially when the batch size is 1000 (F-Storm reduces the latency by 75% compared to Storm). But when the batch size is set to 8000, the latency of F-Storm is higher than the one of Storm, which is due to the high cost of data moving (i.e., more data will be processed and transferred). To demonstrate this point, we test the processing time of FindMatchingWordBolt with different batch sizes and present the results in Figure 11. As expected, the larger the batch size, the longer the processing time is. From these results, we can find that there is a trade-off between the throughput and the tuple processing latency because the larger batch size can achieve higher throughput, but at the same time it results in longer tuple processing time.

In the second experiment, we set the batch size to a fixed value of 4000 and measure three metrics including latency, CPU utilization of the main thread in FindMatchingWordBolt, and the throughput under different input data rates. As illustrated in Figure 10, under a low input data rate of 10000 tuples/sec, F-Storm has a higher latency than Storm and that is due to the longer waiting time in the input buffer. However, when the input data rate is increased to 20K tuples/sec, the latency of F-Storm is much lower than the one of Storm. Even more, when the input data rate is 30K tuples/sec, the latency of Storm increases rapidly with the running time, and finally Storm fails while F-Storm can still keep a relatively stable processing latency. The corresponding throughput of Storm and F-Storm can be seen in Figure 12. When the input data rate is low, Storm and F-Storm can achieve similar throughput. But, when increasing the input data rate, the throughput continues to increase under F-Storm while it stays the same under Storm. These results indicate that F-Storm can deal with higher rate data streams compared to Storm. In addition, Figure 13 shows the result of CPU utilization of the main thread in FindMatchingWordBolt under different input data rates. Although the CPU utilization increases with the input data rate, F-Storm keeps a much lower time compared to Storm. *In summary, we observe that in a small Edge cluster with limited Edge servers, F-Storm can accelerate stream data applications by offloading some operators with heavy load to FPGAs, it can also deal with higher rate data streams compared to Storm.*

D. Vector Addition

Vector addition is one of the fundamental operations of vectors. We design a stream-version topology for it. In this topology, the spout called *VectorGeneratorSpout* generates two vectors, each of which has 500 elements, and then emits them as a tuple. *VectorAddBolt* receives upstream tuples, extracts two vectors, and adds them. Like *FindMatchingWordBolt* in grep, this bolt also can be implemented as a general bolt for Storm or an accBolt for F-Storm. Last, there is still a bolt called *ResultWriterBolt* responsible for writing results to a file.

We set the batch size to 200 and test the throughput under different input data rates. As shown in Figure 14, under low input data rate, Storm and F-Storm have a similar throughput. However, when input data rate is 6K tuples/sec or higher, F-Storm outperforms Storm. When the input rate is 18K tuples/sec, F-Storm improves the throughput by 3.2x compared to Storm: while the throughput of F-Storm is 16730 tuples/sec, Storm achieves a throughput of only 5100 tuples/sec. *This application demonstrates that compared to Storm, F-Storm can deal with heavier data load and achieve higher throughput.*

E. Programming Efforts

The programming work of F-Storm applications can be divided into two parts: topology programming and FPGA accelerator programming. In order to evaluate the programming efforts of F-Storm, we measure the changes of lines of topology code and the additional lines of OpenCL kernel codes for each application. The results are shown in Table I. The second column represents the lines of topology code needed to be modified in the original stream data application in order to access FPGA accelerators, as well as the total lines of topology code. The third column represents the increased lines of code for developing the OpenCL kernel functions. We can observe that using F-Storm programming interface to develop a stream data application which utilizes FPGAs is easy, with less than 20 lines of code of changes in the topology, and few kernel codes for each accelerator. *Hence, F-Storm provides a user-friendly programming interface and can serve various IoT applications in the Edge.*

Table I. LOC (Lines of Code) changes

	Topology (changed/total)	OpenCL Kernel
Matrix Multiplication	14/243	40
Grep	12/265	31
Vector Addition	11/222	11

VII. Discussion

Lessons learned from F-Storm. To the best of our knowledge, F-Storm is the first FPGA-based DSP system which targets Edge servers. Through developing and evaluating F-Storm, we have shown the main requirements and challenges toward efficiently exploiting FPGAs in Edge environments. Beyond that, working with F-Storm helps us to identify other important challenges including: *(1) Decoupling the utilization of FPGA accelerators from specific DSP systems*: It will be useful to build a unified light-weight FPGA accelerator service which can be integrated into different DSP frameworks in the Edge; *(2) Coping with large-scale deployment of Edge clusters*: In Edge computing environment, computing resources (such as Edge servers and FPGAs) may be geographically distributed, how to optimize the operator placement according to the types and the locality of computing resources in order to achieve lower latency is a direction worth studying; *(3) Reducing the burden of users in the development of FPGA enabled applications*: Some common operators are used in many stream data applications, thus it will be useful to develop a library of common stream operator accelerators that can be embedded into stream data applications directly by users.

Use FPGA accelerators or not? FPGAs can exploit spatial and temporal parallelism to accelerate computation-intensive algorithms. But for some simple computation, the kernel speed up achieved by FPGA accelerators is not worth the cost of data transmission between CPUs and FPGAs. Each stream data application may contain several operators and it is important for each operator to decide whether to use FPGA accelerators or not. Letting the users to decide when and where to use FPGAs, is not an ideal method. Future work can provide a smart (online) method to analyze stream data applications in order to identify operators that can benefit from FPGA acceleration service and schedule them accordingly.

Data moving overhead incurred by FPGAs. Reducing data transfer overhead incurred by FPGAs is important for latency-sensitive stream data applications. Besides adopting data batching and data transfer pipelining to solve the problem, it is worth to investigate how to build data transfer paths between multiple FPGA boards: If some tasks on different FPGA boards have to exchange data, they can transfer data directly through the FPGA-FPGA path instead of FPGA-CPU-FPGA path, which can save much data transfer time.

Sharing single FPGA board among multiple accelerators. FPGAs have abundant logic blocks and memory blocks which can provide great computing power to accelerate stream data applications. In our study, a single FPGA board runs only one accelerator which corresponds to one stream operator. However, in the Edge, the stream operators are diverse and some operators accelerated by FPGAs may only use small parts of FPGA resources. If one accelerator monopolizes a single FPGA board, it will result in a waste in the remaining on-chip resources. So it is important to share a single FPGA board among several accelerators in order to maximize the utilization of resources on the board and save the cost of computing resources. However, sharing will introduce several challenges related to interference and performance isolation.

VIII. Conclusion

In this vision paper, we discuss and demonstrate the practical importance of using FPFAs to accelerate stream data applications in Edge servers. We identify several design requirements toward achieving effective stream data processing in the Edge and use them to develop F-Storm, an FPGA-accelerated and general-purpose DSP system for the Edge. We expect this work to offer useful insight into leveraging FPGAs for stream data application in the Edge and to accelerate progress in this domain.

Acknowledgements

This work is supported by National Key Research and Development Program under grant 2016YFB1000501. Shadi's work is partially funded by the ANR KerStream project (ANR-16-CE25-0014-01) and the Stack/Apollo connect talent project.

References

[1] S. Biookaghazadeh, M. Zhao, and F. Ren, "Are fpgas suitable for edge computing?" in *HotEdge'18*, pp. 1–6.

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *SIGMOD'14*, pp. 147–156.

[3] A. Katsifodimos and S. Schelter, "Apache flink: Stream analytics at scale," in *IC2EW'16*, pp. 193–193.

[4] "Spark streaming," http://spark.apache.org/streaming, accessed April 4, 2010.

[5] S. Sakr, Z. Maamar, A. Awad, B. Benatallah, and W. M. P. Van Der Aalst, "Business process analytics and big data systems: A roadmap to bridge the gap," *IEEE Access*, vol. 6, pp. 77 308–77 320, 2018.

[6] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *ICDCS'17*, pp. 2573–2574.

[7] G. Ananthanarayanan, P. Bahl, P. Bodk, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[8] N. Mohamed, J. Al-Jaroodi, I. Jawhar, S. Lazarova-Molnar, and S. Mahmoud, "Smartcityware: A service-oriented middleware for cloud and fog enabled smart city services," *IEEE Access*, vol. 5, pp. 17 576–17 588, 2017.

[9] E. G. Renart, J. Diaz-Montes, and M. Parashar, "Data-driven stream processing at the edge," in *ICFEC'17*, pp. 31–40.

[10] M. Chao, C. Yang, Y. Zeng, and R. Stoleru, "F-mstorm: Feedback-based online distributed mobile stream processing," in *SEC'18*, pp. 273–285.

[11] "Apache edgent," http://edgent.apache.org/.

[12] "Minifi," https://nifi.apache.org/minifi/.

[13] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *SEC'16*, pp. 168–178.

[14] P. Ravindra, A. Khochare, S. Reddy, S. Sharma, P. Varshney, and Y. Simmhan, "ECHO: an adaptive orchestration platform for hybrid dataflows across cloud and edge," *CoRR*, vol. abs/1707.00889, 2017.

[15] S. Wu, M. Liu, S. Ibrahim, H. Jin, L. Gu, F. Chen, and Z. Liu, "Turbostream: Towards low-latency data stream processing," in *ICDCS'18*, pp. 983–993.

[16] M. Najafi, K. Zhang, M. Sadoghi, and H. Jacobsen, "Hardware acceleration landscape for distributed real-time analytics: Virtues and limitations," in *ICDCS'17*, pp. 1938–1948.

[17] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *FPL'16*, pp. 1–10.

[18] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *FCCM'18*, pp. 93–96.

[19] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA'14*, pp. 13–24.

[20] G. Chrysos, P. Dagritzikos, I. Papaefstathiou, and A. Dollas, "Hc-cart: A parallel system implementation of data mining classification and regression tree (cart) algorithm on a multi-fpga system," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 47:1–47:25, 2013.

[21] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *FPGA'17*, pp. 217–226.

[22] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA'16*, pp. 16–25.

[23] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA'15*, pp. 161–170.

[24] A. Nydriotis, P. Malakonakis, N. Pavlakis, G. Chrysos, E. Ioannou, E. Sotiriades, M. N. Garofalakis, and A. Dollas, "Leveraging reconfigurable computing in distributed real-time computation systems," in *EDBT/ICDT Workshops*, 2016, pp. 1–6.

[25] K. Nakamura, A. Hayashi, and H. Matsutani, "An fpga-based low-latency network processing for spark streaming," in *BigData'17*, pp. 2410–2415.

[26] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *SoCC'16*, pp. 456–469.

[27] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[28] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *HotWeb'15*, pp. 73–78.

[29] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.

[30] H. Okuhata, R. Imai, M. Ise, R. Y. Omaki, H. Nakamura, S. Hara, and I. Shirakawa, "Implementation of dynamic-range enhancement and super-resolution algorithms for medical image processing," in *ICCE'14*, pp. 181–184.

[31] Q. Y. Tang and M. A. S. Khalid, "Acceleration of k-means algorithm using altera sdk for opencl," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 1, pp. 6:1–6:19, 2016.

[32] "Opencl overview," http://www.khronos.org/opencl/.

[33] "Intel fpga sdk for opencl," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/ps-opencl.pdf.

[34] Q. Ning, C. Chen, R. Stoleru, and C. Chen, "Mobile storm: Distributed real-time stream processing for mobile clouds," in *CloudNet'15*, pp. 139–145.

[35] J. Morales, E. Rosas, and N. Hidalgo, "Symbiosis: Shar-

ing mobile resources for stream processing," in *ISCC'14*, vol. Workshops, pp. 1–6.

[36] H. Wang and L. Peh, "Mobistreams: A reliable distributed stream processing system for mobile devices," in *IPDPS'14*, pp. 51–60.

[37] R. B. Das, G. D. Bernardo, and H. Bal, "Large scale stream analytics using a resource-constrained edge," in *EDGE'18*, pp. 135–139.

[38] J.-H. Choi, J. Park, H. D. Park, and O.-g. Min, "Dart: Fast and efficient distributed stream processing framework for internet of things," *ETRI Journal*, vol. 39, no. 2, pp. 202–212, 2017.

[39] Y. Xiong, D. Zhuo, S. Moon, M. Xie, I. Ackerman, and Q. Hoole, "Amino - a distributed runtime for applications running dynamically across device, edge and cloud," *SEC'18*, pp. 361–366.

[40] M. Habib ur Rehman, P. P. Jayaraman, S. u. R. Malik, A. u. R. Khan, and M. Medhat Gaber, "Rededge: A novel architecture for big data processing in mobile edge computing environments," *Journal of Sensor and Actuator Networks*, vol. 6, no. 3, pp. 1–17, 2017.

[41] B. Theeten and N. Janssens, "Chive: Bandwidth optimized continuous querying in distributed clouds," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 219–232, 2015.

[42] A. D. S. Veith, M. D. de Assunão, and L. Lefèvre, "Latency-Aware Placement of Data Stream Analytics on Edge Computing," in *ICSOC'18*, pp. 215–229.

[43] G. Amarasinghe, M. D. de Assuno, A. Harwood, and S. Karunasekera, "A data stream processing optimisation framework for edge computing applications," in *ISORC'18*, pp. 91–98.

[44] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for hadoop: Time to rethink?" in *SoCC'13*, pp. 20:1–20:13.

[45] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *FPGA'10*, pp. 93–102.

[46] K. Neshatpour, M. Malik, and H. Homayoun, "Accelerating machine learning kernel in hadoop using fpgas," in *CCGrid'15*, pp. 1151–1154.

[47] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A mapreduce framework on opencl-based fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3547–3560, 2016.

[48] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When apache spark meets fpgas: A case study for next-generation dna sequencing acceleration," in *HotCloud'16*, pp. 64–70.

[49] E. Ghasemi and P. Chow, "Accelerating apache spark with fpgas," *Concurrency and Computation Practice and Experience*, vol. 31, no. 2, p. e4222, 2017.

[50] J. He, Y. Chen, T. Z. J. Fu, X. Long, M. Winslett, L. You, and Z. Zhang, "Haas: Cloud-based real-time data analytics with heterogeneity-aware scheduling," in *ICDCS'18*, pp. 1017–1028.

[51] "Apache hadoop yarn," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[52] "Apache mesos," http://mesos.apache.org/.

[53] J. Cong, P. Wei, and C. H. Yu, "From JVM to FPGA: Bridging abstraction hierarchy via optimized deep pipelining," in *HotCloud'18*, pp. 1–6.

[54] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *EuroSys'13*, pp. 197–210.