Title

Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale

Authors

Huang, Muhuan
Wu, Di
Yu, Cody Hao
et al.

Publication Date

2016-10-05

DOI

10.1145/2987550.2987569

Peer reviewed

HHS Public Access

Author manuscript

Proc ACM Symp Cloud Comput. Author manuscript; available in PMC 2017 March 15.

Published in final edited form as:

Proc ACM Symp Cloud Comput. 2016 October ; 2016: 456–469. doi:10.1145/2987550.2987569.

Programming and Runtime Support to *Blaze* FPGA Accelerator Deployment at Datacenter Scale

Muhuan Huang[1,2,*], **Di Wu**[1,2,*], **Cody Hao Yu**[1,*], **Zhenman Fang**[1], **Matteo Interlandi**[1], **Tyson Condie**[1], and **Jason Cong**[1]

[1]University of California Los Angeles

[2]Falcon Computing Solutions, Inc

Abstract

With the end of CPU core scaling due to dark silicon limitations, customized accelerators on FPGAs have gained increased attention in modern datacenters due to their lower power, high performance and energy efficiency. Evidenced by Microsoft's FPGA deployment in its Bing search engine and Intel's 16.7 billion acquisition of Altera, integrating FPGAs into datacenters is considered one of the most promising approaches to sustain future datacenter growth. However, it is quite challenging for existing big data computing systems—like Apache Spark and Hadoop—to access the performance and energy benefits of FPGA accelerators.

In this paper we design and implement Blaze to provide programming and runtime support for enabling easy and efficient deployments of FPGA accelerators in datacenters. In particular, Blaze abstracts FPGA accelerators as a service (FaaS) and provides a set of clean programming APIs for big data processing applications to easily utilize those accelerators. Our Blaze runtime implements an FaaS framework to efficiently share FPGA accelerators among multiple heterogeneous threads on a single node, and extends Hadoop YARN with accelerator-centric scheduling to efficiently share them among multiple computing tasks in the cluster. Experimental results using four representative big data applications demonstrate that Blaze greatly reduces the programming efforts to access FPGA accelerators in systems like Apache Spark and YARN, and improves the system throughput by 1.7 × to 3× (and energy efficiency by 1.5× to 2.7×) compared to a conventional CPU-only cluster.

Keywords

FPGA-as-a-service; heterogeneous datacenter

Categories and Subject Descriptors

C.1.3 [Computer Systems Organization]; Heterogeneous (hybrid) systems

*Author names are listed in alphabetical order.

1. Introduction

Modern big data processing systems, such as Apache Hadoop [1] and Spark [47], have evolved to an unprecedented scale. As a consequence, cloud service providers, such as Amazon, Google and Microsoft, have expanded their datacenter infrastructures to meet the ever-growing demands for supporting big data applications. However, due to the problem of dark silicon [21], simple CPU core scaling has come to an end, and thus CPU performance and energy efficiency has become one of the primary constraints in scaling such systems. To sustain the continued growth in data and processing methods, cloud providers are seeking new solutions to improve the performance and energy efficiency for their big data workloads.

Among various solutions that harness GPU (graphcs processing unit), FPGA (field-programmable gate array), and ASIC (application-specific integrated circuit) accelerators in a datacenter, the FPGA-enabled datacenter has gained increased attention and is considered one of the most promising approaches. This is because FPGAs provide low power, high energy efficiency and reprogrammability to customize high-performance accelerators. One breakthrough example is that Microsoft has deployed FPGAs into its datacenters to accelerate the Bing search engine with almost 2x throughput improvement while consuming only 10% more power per CPU-FPGA server [33]. Another example is IBM's deployment of FPGAs in its data engine for large NoSQL data stores [13]. Moreover, Intel, with the $16.7 billion acquisition of Altera, is providing closely integrated CPU-FPGA platforms for datacenters [12], and is targeting the production of around 30% of the servers with FPGAs in datacenters by 2020 [6].

With the emerging trend of FPGA-enabled datacenters, one key question is: *How can we easily and efficiently deploy FPGA accelerators into state-of-the-art big data computing systems like Apache Spark* [47] *and Hadoop YARN* [42]*?* To achieve this goal, both programming abstractions and runtime support are needed to make these existing systems programmable to FPGA accelerators. This is challenging for the following reasons.

1. Unlike conventional CPU and GPU targeted programs, compiling an FPGA program can take several hours, which makes existing runtime systems that use dynamic code generation for CPU-GPU datacenters, such as Dandelion [35], HadoopCL [22] and SWAT [23], not applicable for FPGAs.
2. State-of-the-art big data systems like Apache Hadoop and Spark compile to the Java Virtual Machine (JVM), while FPGA accelerators are usually manipulated by C/C++/OpenCL. Even with predesigned FPGA accelerators, there are still excessive programming efforts required to i) integrate them with the JVM, ii) share an accelerator among multiple threads or multiple applications, and iii) share an FPGA platform by multiple accelerators of different functionalities.
3. A straightforward JNI (Java Native Interface) integration of FPGA accelerators can diminish or even degrade the overall performance (up to 1000X slowdown) due to the overwhelming JVM-to-native-to-FPGA communication overhead [15].

4. It usually takes several seconds to reprogram an FPGA into a different accelerator (with a different functionality). A frequent FPGA reprogramming in a multi-accelerator scenario can significantly degrade the overall system performance. This raises a fundamental question: *Do we manage "the hardware platform itself" or "the logical accelerator (functionality) running on top of the hardware platform" as a resource?*

To address these challenges, we design and implement *Blaze*: a framework that provides a programming abstraction and runtime support for easy and efficient FPGA deployments in datacenters. This paper describes the Blaze architecture and makes the following contributions.

1. Programming APIs that enable big data processing applications to leverage FPGA accelerators to perform task-level work. We abstract FPGA accelerators as a service (FaaS), which decouples the hardware accelerator development of data processing tasks (i.e., Spark transformations) and big data processing logic (i.e., scheduling tasks, shuffling data, etc.).[1]
2. Policies for managing logical accelerator functionality —instead of the physical hardware platform itself—as a resource, where better scheduling decisions can be made to optimize the system throughput and energy efficiency.
3. An efficient runtime to share FPGA accelerators in data-centers, where an FaaS framework is implemented to support sharing of accelerators among multiple threads and multiple applications in a single node. Also, an accelerator-centric scheduling is proposed for the global accelerator management to alleviate the FPGA reprogramming overhead for multi-accelerators. Finally several well-known optimization techniques—such as data caching and task pipelining—are employed to reduce the JVM-to-FPGA communication overhead.
4. An open-source prototype that is compatible with existing ecosystems like Apache Spark with no code changes and YARN with a lightweight patch. Our goal is to bring FPGA accelerator developers, big data application developers, and system architects together, to blaze the deployment of accelerators in datacenters.[2]

2. Background

There has been great success in programming frameworks that enable efficient development and deployment of big data applications in conventional datacenters, i.e., composed of general-purpose processors. In this section we briefly introduce Apache Spark [47]— our target big data processing framework— and the Hadoop YARN resource manager [42], which we use to expose FPGA resources in a cluster environment. We also give a quick tutorial of FPGA accelerators.

[1]While Blaze does support GPU accelerators as well, this paper will mainly focus on FPGA accelerators which have not been studied before.

[2]Blaze can be downloaded from github: https://github.com/UCLA-VAST/blaze. Blaze has already been used by multiple groups at Intel Labs to deploy accelerators composed of the Intel-Altera Heterogeneous Accelerator Research Platforms (HARP CPU-FPGA platforms).

2.1 Apache Spark

Apache Spark [47] is a widely used fast and general large-scale data processing framework. It exposes a programming model based on Resilient Distributed Datasets (RDDs) [46]. The RDD abstraction provides *transformations* (e.g., map, reduce, filter, join, etc.) and *actions* (e.g., count, collect) that operate on datasets partitioned over a cluster of nodes. A typical Spark program executes a series of transformations ending with an action that returns a singleton value (e.g., the record count of an RDD) to the Spark driver program, which could then trigger another series of RDD transformations.

Spark caches reused data blocks in memory, often achieving significant performance speedup over the Hadoop MapReduce [1] on iterative applications such as machine learning. Recent studies [9, 32] show that Spark applications are often computation-bound instead of IO or network bound in conventional Hadoop applications. This motivates us to leverage FPGAs to further accelerate the computation.

Spark can be run standalone on a cluster, or with a resource manager like Hadoop YARN [42]. For each Spark application submitted to the YARN cluster, a set of *containers* (see Section 2.2) is gathered from the resource manager matching the available resources and the application con-figuration. For each acquired container, the Spark context launches an *executor*: a JVM instance providing the base runtime for the execution of the actual data-processing computation (i.e., tasks), and managing the application data.

2.2 Apache YARN

YARN (Yet Another Resource Negotiator) is a widely used cluster resource management layer in the Hadoop system that allocates resources, such as CPU and memory, to multiple big data applications (or jobs). Figure 1 shows a high-level view of the YARN architecture. A typical YARN setup would include a single resource manager (RM) and several node managers (NM) installations. Each NM typically manages the resources of a single machine, and periodically reports to the RM, which collects all NM reports and formulates a global view of the cluster resources. The periodic NM reports also provide a basis for monitoring the overall cluster health at the RM, which notifies relevant applications when failures occur.

A YARN job is represented by an application master (AM), which is responsible for orchestrating the job's work on allocated *containers* i.e., a slice of machine resources (some amount of CPU, RAM, disk, etc.). A client submits an AM package—that includes a shell command and any files (i.e., binary executable configurations) needed to execute the command—to the RM, which then selects a single NM to host the AM. The chosen NM creates a shell environment that includes the file resources, and then executes the given shell command. The NM monitors the containers for resource usage and exit status, which the NM includes in its periodic reports to the RM. At runtime, the AM uses an RPC interface to request containers from the RM, and to ask the NMs that host its containers to launch a desired program. Returning to Figure 1, we see the AM instance running with allocated containers executing a job-specific task.

To manage heterogeneous computing resources in the datacenter and provide placement control, YARN recently introduced a mechanism called *label-based scheduling*.

Administrators can specify labels for each server node and expose the label information to applications. The YARN resource manager then schedules the resource to an application only if the node label matches with the application-specified label. Examples of node labels can be an FPGA or GPU, which indicate that the nodes are equipped with a special hardware platform.

2.3 FPGAs

A field-programmable gate array (FPGA) is a reconfigurable integrated circuit with much lower power consumption compared to CPUs and GPUs. Since an FPGA is essentially customizable hardware, it can achieve significant performance speedup despite its low clock frequency. Many factors contribute to the efficiency of FPGA. For example, application-specific computation pipelines can be designed to avoid the conventional instruction fetching and decoding overhead. The data access can also be customized to significantly improve the data reuse. Processing elements of customized computation pipelines can also be duplicated to scale the performance by data parallelism. Because of these techniques and FPGA's energy efficiency, it has been widely adopted in recent years for accelerating the computation-intensive kernels in standalone applications; it achieved 18x to more than 300x kernel speedups [16, 18, 28, 48].

An FPGA implementation is usually based on a hardware description languages (HDL) such as Verilog and VHDL, and it requires a comprehensive knowledge of hardware. Recent development of high-level synthesis (HLS) [19] allows programmers to use a C-based language to design FPGA accelerators. However, the learning-curve for FPGA programming is usually very steep for software programmers, since the optimal implementation still requires a significant amount of FPGA-specific knowledge.

Due to the power wall and dark silicon [21], FPGA acceleration has become increasingly promising, and OpenCL has emerged as a standard framework for FPGA programming. However, there are several fundamental differences between OpenCL applications for FPGA and GPU. Since the architecture of GPU is fixed, GPU programs can be compiled using a just-in-time (JIT) compiler on the fly. FPGAs, on the other hand, are flexible on the architecture level, but require a much longer compilation time (often several hours). This means that an FPGA accelerator has to be generated in advance as a library, and loaded in an OpenCL host program at runtime. Moreover, the OpenCL support for FPGAs is still at an early stage compared to that for GPUs. For example, the Xilinx OpenCL implementation does not support FPGA accelerator sharing by multiple applications. This further motivates our FaaS design for transparent and efficient FPGA accelerator sharing.

3. Blaze System Overview

We design Blaze as a generic system to enable big data applications to easily access FPGA accelerators and implement it as a third-party package that works with existing ecosystems (i.e., Apache Spark and Hadoop YARN), with lightweight changes. Here we give an overview of the Blaze programming and runtime support and discuss how we address the challenges listed in Section 1.

To provide an easy-to-use programming interface, we abstract FPGA accelerators as a service (FaaS) and propose to decouple the software development of big data applications and the hardware development of FPGA accelerators. This means hardware experts can make the best effort to optimize the accelerator design without being burdened with application complexity, and software developers do not need to be aware of tedious hardware details to take advantage of accelerators. Currently, Blaze provides a set of APIs for Spark programs to offload map computations onto accelerators without any change to the Spark framework. All Spark programmers have to do is to register the pre-defined FPGA accelerators (developed by hardware experts) into Blaze as a service, and call the Blaze API to access the customized accelerators. All the accelerator sharing and management logic are transparently handled by our Blaze runtime.

The Blaze runtime system integrates with Hadoop YARN to manage accelerator sharing among multiple applications. As illustrated in Figure 2, Blaze includes two levels of accelerator management. A global accelerator manager (GAM) oversees all the accelerator resources in the cluster and distributes them to various user applications. Node accelerator managers (NAMs) sit on each cluster node and provide transparent accelerator[1] access to a number of heterogeneous threads from multiple applications. After receiving the accelerator computing resources from GAM, the Spark application begins to offload computation to the accelerators through NAM. NAM monitors the accelerator status, handles JVM-to-FPGA data movement and accelerator task scheduling. NAM also performs a heartbeat protocol with GAM to report the latest accelerator status.

We summarize the key features of Blaze as follows.

1. **FPGA accelerators as a service (FaaS)**. The most important role of NAM in Blaze runtime is providing transparent FaaS shared by multiple application jobs (run on the same node) that request accelerators in a fashion similar to software library routines. Each “logical accelerator” library routine exposes a predefined functionality to a Spark program, and can be composed of multiple “physical accelerators” on multiple hardware platforms (e.g., two FPGAs, or one FPGA and one GPU). FaaS automatically manages the task scheduling between logical and physical accelerators. For example, multiple physical accelerators can be allocated for a single logical accelerator for performance-demanding applications, while one physical accelerator can be shared across multiple logical accelerators if each has a low utilization of that physical accelerator.

2. **Accelerator-centric scheduling**. In order to solve the global application placement problem considering the overwhelming FPGA reprogramming overhead, we propose to manage the logical accelerator functionality, instead of the physical hardware itself, as a resource to reduce such reprogramming overhead. We extend the *label-based scheduling* mechanism in YARN to achieve this goal: instead of configuring node labels as ‘FPGA’, we propose to use accelerator functionality (e.g., ‘KMeans-FPGA’, ‘Compression-FPGA’) as node labels. This helps us to differentiate applications that are using the FPGA devices to perform different computations. Therefore, we can delay the scheduling of accelerators with different functionalities onto the same FPGA to avoid

reprogramming as much as possible. Different from the current YARN solution, where node labels are configured into YARN's configuration files, node labels in Blaze are con-figured into NAM through command-line. NAM then reports the accelerator information to GAM through heartbeats, and GAM configures these labels into YARN.

3. **Hiding JVM-to-FPGA communication**. We also employ well-known techniques such as data caching and task pipelining in FaaS to hide the overwhelming JVM-to-native-to-FPGA communication overhead.

4. **Fault tolerance**. The FaaS design in each NAM also helps the fault tolerance of the system. Whenever a fault in the accelerator hardware occurs, NAM can allocate different hardware to fulfill the request, or fallback to CPU execution when no more accelerators are available.

5. **Facilitating rolling upgrades**. FaaS makes it easy to configure heterogeneous accelerator resources on compute nodes in the datacenter, facilitating rolling upgrades of next-generation accelerator hardware and making the system administration of large-scale heterogeneous data-centers more scalable.

In summary, the easy-to-use programming interface, transparent FaaS, and the accelerator-centric scheduling of Blaze makes FPGA accelerator deployment at datacenter scale much easier than existing approaches. Note that the FaaS framework for NAM is provided as a third-party package without any change to Apache Spark, while accelerator-centric scheduling for GAM and NAM is provided as a lightweight patch to Hadoop YARN. In Section 4 and Section 5, we will present more details about the Blaze programming interface and runtime implementation.

4. Blaze Programming Interface

In this section we first describe the programming interfaces of Blaze from two aspects: how to write a big data application that invokes FPGA accelerators, and how to design and register an FPGA accelerator into Blaze. Then we present our support for data serialization during data transfer between JVM and accelerators.

4.1 Application Programming Interface

We implement Blaze as a third-party package that works with the existing Spark framework[3] without any modification of Spark source code. Thus, Blaze is not specific to a particular version of Spark. Moreover, the Blaze programming model for user applications is designed to support accelerators with minimal code changes. To achieve this, we extend the Spark `RDD` to `AccRDD` which supports accelerated transformations. We explain the detailed usage of `AccRDD` in Listing 1 with an example of logistic regression.

[3]Blaze also supports C++ applications with similar interfaces, but we will mainly focus on Spark applications in this paper.

Listing 1

Blaze application example (Spark Scala)

```
val points = sc.textFile(filePath).cache()val train =
blaze.wrap(points)for (i <- 0 until ITERATIONS) {
   bcW = sc.broadcast(weights)
   val gradients = train.map(
         new LogisticAcc(bcW) ).reduce(a + b)
   weights -= gradients}class LogisticAcc(w: Broadcast_var[V])
      extends Accelerator[T, U] {
   val id: String = "LRGradientCompute"
   def call(p: T): U = {
      localGradients.compute(p, w.value)
   }
   ...}
```

In Listing 1, training data samples are loaded from a file and stored to an RDD `points`, and are used to train `weights` by calculating gradients in each iteration. To accelerate the gradient calculation with Blaze, first the RDD `points` needs to be extended to AccRDD `train` by calling the Blaze API `wrap`. Then an accelerator function, `LogisticAcc`, can be passed to the `.map` transformation of the AccRDD. This accelerator function is extended from the Blaze interface `Accelerator` by specifying an accelerator id and an optional `compute` function for the fall-back CPU execution. The accelerator id specifies the desired accelerator service, which in the example is “LRGradient-Compute”. The fall-back CPU function will be called when the accelerator service is not available. This interface is provided with fault-tolerance and portability considerations. In addition, Blaze also supports caching for Spark broadcast variables to reduce JVM-to-FPGA data transfer. This will be elaborated in Section 5.3.

The application interface of Blaze can be used by library developers as well. For example, Spark MLlib developers can include Blaze-compatible codes to provide acceleration capabilities to end users. With Blaze, such capabilities are independent of the execution platform. When accelerators are not available, the same computation will be performed on CPU. In this case, accelerators will be totally transparent to the end users. In our evaluation, we created several implementations for Spark MLlib algorithms such as logistic regression and K-Means using this approach.

4.2 Accelerator Programming Interface

For accelerator designers, the programming experience is decoupled with any application-specific details. An example of the interface implementing the “LRGradientCompute” accelerator in the prior subsection is shown in Listing 2.

Listing 2

Blaze accelerator example (C++)

```
class LogisticTask : public Task {public:
    LogisticTask(): Task(NUM_ARGS)
```

```
// overwrite the compute function
virtual void compute() {
   int num_elements = getInputLength(...);
   double *in = (float*)getInput(...);
   double *out = (float*)getOutput(...);
   // perform computation
   ...
}};
```

Our accelerator interface hides details of FPGA accelerator initialization and data transfer by providing a set of APIs. In this implementation, for example, the user inherits the provided template, `Task`, and the input and output data can be obtained by simply calling `getInput` and `getOutput` APIs. No explicitly OpenCL buffer manipulation is necessary for users. The runtime system will prepare the input data and schedule it to the corresponding task. The accelerator designer can use any available programming framework to implement an accelerator task as long as it can be integrated with an interface in C++.

4.3 Serialization Support

The input and output data of Spark tasks need to be serialized and deserialized respectively before they are transferred to and from accelerator platforms. Blaze implementation includes its own (de)serializer for primitive data types, because the existing Java version is not sufficient for handling the data layout for accelerators. In addition, Blaze also provides an interface to users to implement their own (de)serializer methods. As a result, users are allowed to use arbitrary data types in the Spark application as long as the corresponding (de)serializer is able to process data to match the accelerator interface.

5. Blaze Runtime Support

In this section, we present our Blaze runtime support, including the FaaS implementation to share accelerators among multiple heterogeneous threads in a single node, accelerator-centric scheduling to alleviate the FPGA reprogramming overhead, communication optimization to alleviate the JVM-to-FPGA overhead, and fault tolerance and security support.

5.1 FPGA-as-a-Service (FaaS)

Blaze facilitates FaaS in NAM through two levels of queues: *task queues* and *platform queues*. The architecture of NAM is illustrated in Figure 3. Each *task queue* is associated with a “logical accelerator”, which represents an accelerator library routine. When an application task requests a specific accelerator routine, the request is put into the corresponding *task queue*. Each *platform queue* is associated with a “physical accelerator”, which represents an accelerator hardware platform such as an FPGA board. The tasks in *task queue* can be executed by different *platform queues* depending on the availability of the implementations. For example, if both GPU and FPGA implementations of the same accelerator library routine are available, the task of that routine can be executed on both devices.

This mechanism is designed with three considerations: 1) application-level accelerator sharing, 2) minimizing FPGA reprogramming, and 3) efficient overlapping of data transfer and accelerator execution to alleviate JVM-to-FPGA over- head. We elaborate the first two considerations in the rest of this subsection, and discuss 3) in Section 5.3.

In Blaze, accelerator devices are owned by NAM rather than individual applications. The reasoning behind this design is our observations that in most big data applications, the accelerator utilization is less than 50%. If the accelerator is owned by a specific application, then much of the time it will be spent in idle, wasting energy. The application-level sharing inside NAM is managed by a scheduler that sits between application requests and *task queues*. In this paper, a simple first-come-first-serve scheduling policy is implemented. We leave the exploration of different policies to future work.

The downside of providing application sharing is the additional overheads of data transfer between the application process and NAM process. For latency-sensitive applications, Blaze also offers a reservation mode where the accelerator device is reserved for a single application, i.e., a NAM instance will be launched inside the application process.

The design of the *platform queue* focuses on mitigating the large overhead in FPGA reprogramming. For a processor-based accelerator such as GPU to begin executing a different "logical accelerator", it simply means loading another program binary, which incurs minimum overhead. With FPGA, on the other hand, the reprogramming takes much longer. An FPGA device contains an array of logic cells, and the programming is effectively configuring the logic function and connection of each cell. Each con-figuration is called a "bitstream", and it typically takes 1~2 seconds to program an FPGA with a given bitstream. Such a reprogramming overhead makes it impractical to use the same scheme as the GPU in the runtime system. In Blaze, a second scheduler sits between *task queues* and *platform queues* to avoid frequent reprogramming of the same FPGA device. More details about the scheduling policy will be presented in the next subsection.

5.2 Accelerator-centric Scheduling

In order to mitigate the FPGA reprogramming overhead, it is better to group the tasks that need the same accelerator to the same set of nodes. The ideal situation is that each cluster node only gets the tasks that are requesting the same accelerator, in which case FPGA reprogramming is not needed. Figure 4 illustrates that grouping accelerator tasks can reduce FPGA reprogramming overhead.

By managing logical accelerator functionality as a resource, we propose an accelerator-locality-based delay scheduling policy to dynamically partition the cluster at runtime, avoiding launching mixed FPGA workloads on the same cluster node as much as possible. During accelerator allocation in GAM, we consider the nodes in the following order as scheduling priorities: 1) the idle nodes that do not have any running containers; 2) the nodes that run similar workloads; 3) the nodes that run a different set of workloads. Specifically, we define an affinity function to describe ith node's affinity to an application as $f_i = \frac{n_{acc}}{n}$, where n_{acc} is the number of containers on this node that use the same logical accelerator (or label), and n is the total number of containers on this node. A node with higher affinity

represents a better scheduling candidate. An idle node which has zero running containers has the highest affinity and is considered the best scheduling candidate. GAM tries to honor nodes with higher accelerator affinity by using the so-called delay scheduling.

At runtime, each NAM periodically sends a heartbeat to the GAM, which represents a scheduling opportunity. The GAM scheduler does not simply use the first scheduling opportunity it receives. Instead, it may skip a few scheduling opportunities and wait a short amount of time for a scheduling opportunity with a better accelerator affinity. In our implementation, we maintain a threshold function for each application, which linearly decreases as the number of missed scheduling opportunities increases. A container is allocated on a node only if the node's accelerator affinity is higher than the threshold function.

5.3 Hiding JVM-to-FPGA Communication

In order for a Spark program to transfer data to an FPGA accelerator, the data has to be first moved from JVM to the native machine, and then moved to the FPGA device memory through a PCIe connection. Such data movement between the host CPU and FPGA accelerators sometimes can diminish or even degrade the overall system performance [15]. To mitigate such overhead, Blaze adopts the following well-known techniques within the FaaS framework.

1. **Task pipelining**. Most datacenter workloads will have multiple threads/tasks sharing the same accelerator, which creates an opportunity to hide data transfer with task execution by pipelining: the *task queue* in NAM adopts an asynchronous communication scheme that overlaps JVM-to-FPGA data communication with FPGA accelerator execution.

2. **FPGA data caching**. Many big data applications like machine learning use iterative algorithms that repeatedly perform computation on the same set of input data. This provides the opportunity to cache the data on the FPGA device memory and thus avoid the most time-consuming native-to-FPGA data movement through PCIe. To be more specific, our FaaS framework implements a *Block Manager* to maintain a data reuse table that records the mapping from the native data block to the FPGA device memory block. For the case of OpenCL, *Block Manager* manages a table of `cl buffer` objects which are mapped to device memory. A flag is used to indicate whether the programmer wants Blaze to cache an input data block. In Spark, the flag is automatically assigned if the user specifies **.cache()** for the input RDD.

3. **Broadcast data caching**. Most data analytic frameworks such as Spark support data sharing across the cluster nodes. In Spark, this is provided as broadcast data. Similarly, Blaze also supports a broadcast data caching to minimize data transfer across the cluster nodes. A broadcast block only needs to be transferred to the NAM once, and it will be cached inside the *Block Manager* throughout the application's life cycle.

5.4 Fault Tolerance and Security Issues

Fault tolerance is inherent in our proposed transparent accelerator scheduling. All accelerator-related errors are caught at the application level, and the CPU implementation will be used to resume the execution. Errors of accelerators in NAM are handled in a similar fashion as Spark or YARN. A counter is used for each *accelerator task* per platform, keeping track of the number of errors incurred. If the failure is persistent for one *accelerator task*, it will be removed from NAM's configuration. This information will also be propagated to GAM in the heartbeat signals, and GAM will remove the corresponding label for this node.

Based on the description of the Blaze accelerator interface in Section 4.2, the accelerator task implementation only has access to its private input data through the provided interface, such as `getInput()`. The data can only be assigned by NAM based on the dependency, and all input data is read-only. Our underlying platform implementation is based on existing accelerator runtime systems such as OpenCL, so we rely on the runtime implementation to guarantee security at the device level. In general, the security issues in FPGA-enabled datacenters will be an open and interesting direction for future work.

6. Experimental Results

Now we evaluate the programming efforts and system performance of deploying FPGA accelerators in datacenters using Blaze. First we present the hardware and software setup, and describe the four representative large-scale applications we chose that cover two extremes: iterative algorithms like machine learning, and streaming algorithms like compression and genome sequencing. We evaluate the programming efforts to write these applications using Blaze in terms of lines-of-code (LOC). Then we evaluate the overall system speedup and energy savings for each individual application by putting FPGA accelerators into the cluster. We also analyze the FaaS overhead and break down the performance improvement of each optimization. Finally, we analyze multi-job executions and the efficiency of our accelerator-centric scheduling policy in the global accelerator management.

6.1 Experimental Setup

The experimental platform we use is a local standard CPU cluster with up to 20 nodes, among which 4 nodes[4] are integrated with FPGA cards using PCI-E slots. Each server has dual-socket Intel Xeon E5-2620v3 CPUs with 12 cores in total and 64GB of main memory. The FPGA card is AlphaData ADM-PCIE-7V3, which contains a Xilinx Virtex-7 XC7VX690T-2 FPGA chip and 16GB of on-board DDR3 memory. The FPGA board can be powered by PCI-E alone and consumes around 25W, which makes it deployable into commodity datacenters.

The software framework is based on a community version of Spark 1.5.1 and Hadoop 2.6.0. The accelerator compilation and runtime are provided by the vendor toolkits. For the

[4]We are planning to install more FPGA cards in the near future.

AlphaData FPGA cards, we use the OpenCL flow provided by the Xilinx SDAccel tool-chain, where the OpenCL kernels will be synthesized into bitstreams to program the FPGA.

We choose a set of four representative compute-intensive large-scale applications. They cover two extremes: iterative machine learning algorithms like logistic regression and K-means clustering, and streaming algorithms like genome sequencing analysis and Apache Parquet compression.

1. **Logistic regression (LR)**. The baseline LR is the training application implemented by Spark MLlib [11] with the LBFGS algorithm. The software baseline uses netlib with native BLAS library. The computation kernels we select are the logistic gradients and the loss function calculation. The kernel computation takes about 80% of the total application time.

2. **K-Means clustering (KM)**. The KM application is also implemented using Spark MLlib, which uses netlib with native BLAS library. The computation kernel we select is the local sum of center distances calculation. The datasets used in KM are the same as LR, and the percentage of kernel computation time is also similar to LR.

3. **Genome sequences alignment (GSA)**. The GSA application is from the open-source Cloud Scale BWAMEM (CS-BWAMEM) software suite [17], which is a scale-out implementation of the BWAMEM algorithm [29] widely used in the bioinformatics area. The algorithm aligns the short reads from the sequencer to a reference genome. We mainly focus on the alignment step in this application which uses the Smith-Waterman algorithm, as we did in a prior case study [15].

4. **Apache Parquet compression (COMP)**. Apache Parquet [2] is a compressed and efficient columnar data representation available to any project in the Hadoop/Spark ecosystem. Such columnar data generally have good compression rates and thus are often compressed for better spatial utilization and less data communication. We mainly focus on the compression (deflater) step, which is computation-bound and common through various applications. We use two software baselines: 1) the Java Gzip implementation that uses both the LZ77 algorithm and Huffman encoding, which has a better compression ratio but low throughput; and 2) the open-source Snappy implementation [10] that uses a JNI wrapper to call the C++ Snappy library based on the LZ77 algorithm, which has a lower compression ratio but better throughput.

The input data for LR and KM are based on a variant of the MNIST dataset [8] with 8 million records, and is sampled such that on average each node will process 2–4GB of data. The data set of GSA is a sample of HCC1954, which is a single person's whole genome. The input data for COMP is the first 100 kilo short reads in HCC1954.

The FPGA accelerators for all applications are designed in-house. The accelerator specifications for LR and KM can be found in [18], and the Smith-Waterman implementation is based on [16]. Our FPGA accelerator is designed based on the Gzip implementation with both the LZ77 algorithm and Huffman encoding. Table 1 presents an

overview of the accelerator speedup compared to the 12-thread CPU software baseline in terms of throughput improvement. We set `--num-executors` to 1 and `--executor-cores` to 12 in Spark. For COMP, 12 independent streams on the CPU are used to take advantage of all cores. The accelerator design details are omitted in this paper, since our focus is on evaluating the integration benefits of FPGA accelerators into big data applications using Blaze.

Currently, we only run the kernel computation on FPGAs for FPGA-related experiments as a proof-of-concept. We will consider efficient CPU-FPGA co-working in our future work, which will provide higher performance than our current reported results.

6.2 Programming Efforts

We begin the analysis by showing Blaze's benefits in reducing the deployment efforts of integrating existing FPGA accelerators to big data applications. The results are shown in Table 2, where the lines of code (LOC) breakdown is listed for the selected applications. The hardware code to design the accelerators is exactly the same between manual and Blaze implementations and decoupled from software developers, so it is excluded in this comparison. As an illustration of complexity of accelerator designs, it usually takes an experienced hardware engineer around 4 to 24 weeks to implement an efficient FPGA accelerator kernel, which is a big overhead for big data application developers like Spark programmers. In this paper the LR, KM, GSA, and COMP accelerators take a senior graduate student 4, 4, 24, and 16 weeks to implement and optimize. Column 'App' in Table 2 shows code changes needed to modify the big data applications so as to access accelerators in the application code. Column 'ACC-setup' shows the code changes for PCIe data transfer and accelerator invocation through OpenCL. Finally, column 'Partial FaaS' shows the code changes needed to enable sharing accelerators among multiple threads within the application.

Although using LOC to represent the programming efforts is not entirely accurate, it provides a rough illustration of the difference between each implementation method. Among the breakdown of LOCs, most of the "ACC-setup" code for accelerator control can be reused as long as the accelerator is fixed. We can see that deploying FPGA accelerators in big data applications using Blaze is very easy, with less than 10 LOC changes in the application, and a one-time 100 LOC changes for accelerator setup. Without Blaze, even a manual design for partial FaaS to support accelerator sharing among multi-threads within a single application requires 325 to 896 LOC changes for every application.

6.3 Overall System Performance and Energy Gains for Single Application

Figure 5 demonstrates the single-node system speedup[5] and energy reduction for our application case studies using Blaze and FPGA accelerators. For each individual job, we measure the overall job time and estimate the overall energy consumption based on the average power measured during application runtime. As mentioned earlier, we only run the

[5]For Figure 5, 6, and 7, the experiments are done by configuring `--executor-cores` to 12 and `--num-executors` to the number of nodes in Spark.

kernel computation on FPGAs. Compared with the CPU baseline, the system with FPGA achieved 1.7× to 3× speedup on overall system throughput, and 1.5× to 2.7× improvement on system energy consumption. (Note that FPGAs introduce an additional 25 watts per node into the system; therefore the achieved energy efficiency is slightly smaller than the performance speedup numbers.) This confirms that computation-intensive big data applications can take full advantage of FPGA acceleration with Blaze.

Moreover, we compare the performance of a 4-node cluster with FPGAs to the CPU-only clusters with 4-node, 8-node, and 12-node. As shown in Figure 6, for LR and KM, a 4-node cluster with FPGA accelerators enabled can provide roughly the same throughput as a cluster of 12 CPU nodes. This indicates that we can reduce the conventional datacenter size by 3× by putting an FPGA into each server node, while achieving the same throughput.

Finally, Figure 7 presents the execution time breakdown of Spark jobs (the entire application instead of the kernel task execution time) on a 4-node cluster before and after FPGA acceleration. The results confirm that machine learning workloads such as LR and KM are computationally intensive, and the computation kernels benefit from FPGA acceleration. Note that the data load and preprocessing part in the original Spark program remain on the CPU, i.e., it is not accelerated by FPGA.

6.4 FaaS Overhead Analysis

To evaluate the potential overhead that Blaze introduces to provide FaaS, we evaluate the performance of Blaze integration against a reference manual integration. To make the analysis simple, we focus on the streaming COMP application. We first measure the normalized compression throughput to the reference manual design for 1-core and 12-core cases. As shown in Figure 8(a), for the two software baselines, the native Snappy implementation is around 10× faster than the Java Gzip implementation. For the single-core version, a manual integration of the compression FPGA accelerator achieves around 8.5× speedup over Snappy, while a Blaze integration achieves around 5.6× speedup. When there are 12 cores, the fully parallelized software implementation gets significant speedup, while Blaze integration and manual integration achieve similar performance, which is 1.7× better than Snappy.

Then we analyze why Blaze integration has more overhead than manual integration in the single-core case. We break down the execution time into FPGA kernel execution, JVM-to-native and native-to-FPGA data movement, and private-to-shared memory movement in Blaze native. The detailed breakdown is illustrated in Figure 8(b). As we can see, Blaze introduces the overhead of moving data from application private memory to the Blaze shared memory, which is required to manage accelerator sharing by multiple applications and costs around 50% more execution time. Figure 8(b) also confirms that the overwhelming JVM-to-FPGA communication overhead occupies 76% of the total execution time in the single-core COMP application. Due to the multi-thread nature in big data applications, such overhead can be alleviated using task pipelining (and data caching) that is transparently supported by our FaaS framework in Blaze. As a result, we see a comparable performance between Blaze integration and manual integration when there are 12 cores.

6.5 Breakdown of FaaS Optimizations

We show the breakdown of performance improvements by each JVM-to-FPGA communication optimization in Figure 9. We start from a naive FaaS without task pipelining or data caching, and then gradually add task pipelining and data caching. For each FaaS setup, we evaluate the FPGA kernel time and the task time. The task time represents the targeted accelerating kernel instead of the entire application, which includes both the time of data transfer to and from FPGA via PCIe and FPGA kernel time. FPGA kernel time stays the same across different cases since the total computation that needs to be performed on the FPGA remains the same.

As shown in Figure 9, a naive offloading of workload to accelerator may result in a slow-down rather than a speedup, e.g., 6.71× slowdown for LR and 6.95× slowdown for KM, due to the aforementioned JVM-to-FPGA overhead. By enabling data pipelining, the total time can be accelerated by a factor of 2.8× to 3.8×. For iterative computation of LR and KM, data caching provides a huge performance improvement since most of the data transfer is mitigated. Since all the data in GSA and COMP is processed only once, the results with and without data caching are identical, and thus omitted in Figure 9.

The benefits of task pipelining and data caching can be better illustrated using the accelerator utilization metric. In Figure 10 we show the different utilization patterns of running a single application LR on an FPGA. The accelerator utilization is defined as the ratio of accelerator execution time in a sampled interval of application execution time. The accelerator utilization is consistently low in the case without caching or pipelining shown in the first part of the figure, since the accelerator keeps waiting for data to be transferred from the application. In the second part, when pipelining is enabled, the accelerator can reach high utilization periodically. This is because at the beginning of each iteration the first batch of data needs to be transferred before the accelerator can start, but once the pipeline begins, the accelerator can be kept busy with data continuously flowing in. Once data caching is enabled, the accelerator utilization can be increased dramatically. Similar results can also be observed for KM workloads as well. The high accelerator utilization in full-featured FaaS for KM and LR applications confirms again that the Blaze runtime overhead is negligible.

6.6 Multi-Job Scheduling Analysis

To evaluate the effectiveness of GAM's resource allocation policy (i.e., accelerator-centric scheduling), we choose seven sets of workloads on a 4-node CPU-FPGA cluster. Each set contains LR and KM applications of various input data sizes, and the ratio of these two applications varies among different sets of workloads.

We compare GAM with two baselines: *static-partition* and *naive sharing*. In *static partition*, we evenly partition the 4 nodes into two sets: 2 nodes only run LR applications and the other 2 nodes only run KM applications. Therefore, reprogramming never occurs in the experiments. In *naive sharing*, all the FPGA nodes can run both LR and KM workloads, and we use the Apache YARN's default resource allocation policy. Our GAM has settings similar to *naive sharing*, but uses our accelerator-centric scheduling policy. We also calculate

the offline theoretical optimal scheduling results, in which case we assume that all the sets of workloads submitted are known beforehand.

Figure 11 plots the normalized system throughput to theoretical optimal and accelerator utilization. Comparing the baseline *static partition* with *naive sharing*, we find that static partition performs better when the cluster is partitioned in a way that the ratio of KM nodes to LR nodes is close to the ratio of KM workloads to LR workloads (*i.e.,* ratio is 0.5), while *naive sharing* performs better when the workloads only contain LR or KM applications (*i.e.,* ratio is 1 or 0), since the applications can use all 4 FPGA nodes. However the advantages of *naive sharing* decline as the workloads become more heterogeneous due to FPGA reprogramming overhead.

GAM incorporates the best aspects of *static partition* and *naive sharing*: it potentially allows applications to use all cluster FPGAs (shown as the accelerator utilization rate in Figure 11 (b)). Meanwhile, it reduces FPGA reprogramming overhead by placing similar workloads on the same set of nodes. On average, *static partition* and *naive sharing* are 27% and 22% away from the theoretical optimal results, while GAM is only 9% away from the optimal results.

7. Related Work

There are several projects on the inclusion of heterogeneous architectures in big-data analytic frameworks. In this section we first discuss the projects that manage large-scale clusters and their support for accelerators. Then we review existing runtime systems that were designed and implemented for CPU-GPU datacenters. As we mentioned in Section 1, the approaches for GPUs are almost not applicable to FPGAs. Finally, we consider existing systems especially designed for FPGAs.

Cloud-scale resource management

Resource managers have a long history, and are widely used in managing datacenter-scale clusters of machines. Examples include virtual machine provisioning software, systems that provision long-running services, and scientific cluster management for workloads such as MPI and HTCondor [5]. The most fundamental difference between these systems and resource managers such as Hadoop YARN, is that YARN specifically targets data processing jobs, which elastically request leases on transient resources, returning those resources when the job completes.[6] Such jobs must be written with the assumption that resources can be preempted or fail, and save partial state as required to avoid recomputation. Resource managers that are similar to YARN include Mesos [24], Omega [37] and Corona [4]. However, none of these yet provide support for FPGA accelerator management.

Distributed runtime systems for GPUs

There are several works on managing GPUs at cluster scale. Yahoo [7] demonstrates running deep learning jobs onto a cluster of GPU nodes managed by YARN. Their system leverages YARN's node label capabilities to allow jobs to state whether they should be launched on CPU or GPU nodes. Dandelion [35] uses Moxie, a cluster dataflow engine, to schedule jobs

[6]This is the intended use case for YARN. However, not all big data systems follow this design principle.

represented in a dataflow graph onto a cluster of powerful machines with GPUs. The high-level architecture of Moxie is similar to Dryad [26] and YARN. Although these GPU management techniques can be used to manage FPGA systems, they cannot achieve the same efficiency as our Blaze because they lack consideration of FPGA reprogramming overhead.

There are also some projects that attempt to include GPUs into single-application runtime systems. Caffe [27] is a C-based distributed system for convolutional architecture acceleration on GPU. HeteroSpark [30] is a CPU-GPU Spark framework. Both of them adopt fixed functions, so users have to use provided accelerators to describe their applications. On the other hand, since the compilation time for GPU is negligible, more recent works attempt to provide a fully programmable framework which is able to generate and compile GPU kernel code on the fly. For example, MapCG [25], GPMR [40], and Glasswing [20] allow users to write their own *map/reduce* functions, but the programming models they provided still leverage low-level hardware architecture-aware APIs such as thread ID. In addition, HeteroDoop [36], HadoopCL [22], and SWAT [23] propose a user-friendly programming model for Hadoop and Spark users to write *map/reduce* functions without considering the underlying architecture, but sacrifice some performance improvement opportunities.

Distributed runtime systems for FPGAs

There are some research projects that try to accelerate big-data analytic frameworks using FPGAs. FPMR [39] attempts to implement an entire MapReduce framework on FPGAs so that the data communication overhead can be eliminated. However FPMR still requires users to write customized *map/reduce* functions in RTL, and it only supports very limited MapReduce features. Axel [41] and [44] are C-based MapReduce frameworks for not only FPGAs but also GPUs. Both frameworks have straightforward scheduling mechanisms to allocate tasks to either FPGAs or GPUs. Melia [43] presents a C++ based MapReduce framework on OpenCL-based Altera FPGAs. It generates Altera FPGA accelerators from user-written C/C++ *map/reduce* functions and executes them on FPGAs along with CPUs. Although experimental results in [43] evaluate the quality of generated accelerators, the results for multi-node clusters only come from simulations instead of end-to-end evaluations. Since all of the above systems are implemented in C/C++ as standalone frameworks, they are not compatible with widely used JVM-based big data analytic systems such as Hadoop and Spark.

On the other hand, to bridge the gap between JVM-based frameworks and C-based FPGA accelerators, [45] deploys Hadoop on a server with NetFPGAs connected via an Eth-ernet switch. However, the programming model in [45] exposes low-level interactions with the FPGA device to users. In addition, no results are provided to show whether this system has reasonable scalability when extending to a cluster of multiple servers. Different from [45], the Zynq-based cluster [31] deploys Hadoop on a cluster of Xilinx Zynq SoC devices [34] where CPU cores and programmable logics are fabricated on the same chip. Although the system is energy efficient because of FPGAs, this methodology is tightly coupled with the underlying Zynq platform and is hard to port to clusters with commodity CPU servers. One

of the first studies that integrates PCIe-based high-performance FPGA accelerators into Spark running on commodity clusters is our case study for genome sequence alignment in [15]. Compared to [15], Blaze provides generic programming and runtime support to deploy any FPGA accelerators, and manages accelerator sharing by multiple applications.

FPGA virtualization has been discussed in [14], where multiple FPGA accelerators that are designed with certain coding templates can be placed on the same FPGA board and managed through OpenStack. Compared to our work, this mechanism imposes a limited programming model for accelerator designs due to the use of coding templates. Moreover dividing the FPGA resources into several regions results in less logics being available to accelerators, and it thus limits the performance of accelerators.

SparkCL [38] works in a direction that is orthogonal to our Blaze system described in this paper. While Blaze assumes predefined FPGA accelerators, SparkCL adapts Aparapi [3] to automatically generate an OpenCL kernel for Altera FPGAs and executes FPGA accelerators on Spark. However, the programming model of SparkCL discloses low-level OpenCL APIs such as thread ID to users and only supports primitive data types. In addition, it lacks experimental results to illustrate system efficiency and scalability. We have another ongoing effort at UCLA to improve automatic Java-to-FPGA code generation, which is orthogonal to Blaze and will be integrated into Blaze in future work.

In summary, to the best of our knowledge, Blaze is the first (open source) system that provides easy and efficient access of FPGA accelerators for big data applications that run on top of Apache Spark and Hadoop YARN.

8. Conclusion

In this paper we present the design and implementation of Blaze, which provides programming and runtime support that enables rapid and efficient deployment of FPGA accelerators at warehouse-scale. Blaze abstracts FPGA accelerators as a service (FaaS), decouples the FPGA accelerator development and big data application development, and provides a set of clean programming APIs for big data applications to easily access the performance and energy gains of FPGA accelerators. In the FaaS framework, we provide efficient accelerator sharing by multiple heterogeneous threads, hide the overwhelming Java-to-FPGA data communication overhead, and support fault tolerance. We implement FaaS as a third-party package that works with Apache Spark. In addition, we propose to manage the logical accelerator functionality as a resource instead of the physical hardware platform itself. Using this new concept, we are able to extend Hadoop YARN with an accelerator-centric scheduling policy that better manages global accelerator resources and mitigates the FPGA reprogramming overhead. Our experiments with four representative big data applications demonstrate that Blaze greatly reduces the programming efforts, and improves the system throughput from 1.7× to 3×, i.e., a 1.7× to 3× datacenter size reduction using FPGAs with the same throughput. We also demonstrate that our FaaS implementation achieves performance similar to a manual design under the dominant multi-thread scenarios in big data applications, while our accelerator-centric scheduling achieves close to optimal system throughput.

Acknowledgments

This work is partially supported by the Center for Domain-Specific Computing under the NSF InTrans Award CCF-1436827, funding from CDSC industrial partners including Baidu, Fujitsu Labs, Google, Huawei, Intel, IBM Research Almaden, and Mentor Graphics; C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; grants NSF IIS-1302698 and CNS-1351047; and U54EB020404 awarded by NIH Big Data to Knowledge (BD2K).

References

1. [Accessed: 2016-05-24] Apache Hadoop. https://hadoop.apache.org
2. Apache parquet. [Accessed: 2016-05-24] https://parquet.apache.org/
3. [Accessed: 2016-05-24] Aparapi in amd developer website. http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/
4. [Accessed: 2016-01-30] Facebook engineering (2012) under the hood: Scheduling mapreduce jobs more efficiently with corona. https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920
5. [Accessed: 2016-05-24] HTCondor. https://research.cs.wisc.edu/htcondor
6. [Accessed: 2016-05-17] Intel to Start Shipping Xeons With FPGAs in Early 2016. http://www.eweek.com/servers/intel-to-start-shipping-xeons-with-fpgas-in-early-2016.html
7. [Accessed: 2016-05-24] Large scale distributed deep learning on Hadoop clusters. http://yahoohadoop.tumblr.com/post/129872361846/large-scale-distributed-deep-learning-on-hadoop
8. [Accessed: 2016-05-24] The MNIST database of handwritten digits. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist8m
9. [Accessed: 2016-08-10] Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
10. [Accessed: 2016-08-01] The snappy-java port. https://github.com/xerial/snappy-java
11. [Accessed: 2016-05-24] Spark MLlib. http://spark.apache.org/mllib/
12. [Accessed: 2016-05-17] Xeon+FPGA Platform for the Data Center. https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf
13. Brech, B., Rubio, J., Hollinger, M. Tech rep. IBM Systems Group; 2015. IBM Data Engine for NoSQL - Power Systems Edition.
14. Byma, S., Steffan, JG., Bannazadeh, H., Garcia, AL., Chow, P. FPGAs in the cloud: Booting virtualized hardware accelerators with openstack. Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on; IEEE; 2014. p. 109-116.
15. Chen, Y-T., Cong, J., Fang, Z., Lei, J., Wei, P. When Apache Spark meets FPGAs: A case study for next-generation dna sequencing acceleration. The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16); 2016.
16. Chen, YT., Cong, J., Lei, J., Wei, P. A novel high-throughput acceleration engine for read alignment. Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on; May 2015; p. 199-202.
17. Chen Y-T, Cong J, Li S, Peto M, Spellman P, Wei P, Zhou P. CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing. High Throughput Sequencing Algorithms and Applications (HITSEQ). 2015
18. Cong, J., Huang, M., Wu, D., Yu, CH. Heterogeneous datacenters: Options and opportunities. Proceedings of the 53nd Annual Design Automation Conference; ACM; 2016.
19. Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z. High-level synthesis for FPGAs: From prototyping to deployment. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 30. Apr.2011 4:473–491.
20. El-Helw, I., Hofman, R., Bal, HE. Glasswing: Accelerating mapreduce on multi-core and many-core clusters. Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing; New York, NY, USA. ACM; 2014. p. 295-298.HPDC '14

21. Esmaeilzadeh, H., Blem, E., St Amant, R., Sankar-alingam, K., Burger, D. Dark silicon and the end of multicore scaling. Computer Architecture (ISCA), 2011 38th Annual International SYMPOSIUM on; June 2011; p. 365-376.
22. Grossman, M., Breternitz, M., Sarkar, V. HadoopCL: Mapreduce on distributed heterogeneous platforms through seamless integration of Hadoop and OpenCL. Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum; Washington, DC, USA. IEEE Computer Society; 2013. p. 1918-1927.IPDPSW '13
23. Grossman, M., Sarkar, V. Swat: A programmable, in-memory, distributed, high-performance computing platform. The 25th International Symposium on High-Performance Parallel and Distributed Computing (HPDC); 2016.
24. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, AD., Katz, R., Shenker, S., Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation; Berkeley, CA, USA. USENIX Association; 2011. p. 295-308.NSDI'11
25. Hong, C., Chen, D., Chen, W., Zheng, W., Lin, H. MapCG: Writing parallel program portable between CPU and GPU. Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques; New York, NY, USA. ACM; 2010. p. 217-226.PACT '10
26. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D. ACM SIGOPS Operating Systems Review. Vol. 41. ACM; 2007. Dryad: distributed data-parallel programs from sequential building blocks. In; p. 59-72.
27. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. 2014 arXiv preprint arXiv:1408.5093.
28. Choi, kY, Cong, J. Acceleration of EM-based 3D CT reconstruction using FPGA. IEEE Transactions on Biomedical Circuits and Systems 10. Jun.2016 3:754–767.
29. Li H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. 2013 arXiv preprint arXiv:1303.3997.
30. Li, P., Luo, Y., Zhang, N., Cao, Y. HeteroSpark: A heterogeneous CPU/GPU spark platform for machine learning algorithms. Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on; Aug 2015; p. 347-348.
31. Lin, Z., Chow, P. Zcluster: A Zynq-based Hadoop cluster. Field-Programmable Technology (FPT), 2013 International Conference on; Dec 2013; p. 450-453.
32. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B-G. Making sense of performance in data analytics frameworks. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15); Oakland, CA. May 2015; USENIX Association; p. 293-307.
33. Putnam, A., Caulfield, AM., Chung, ES., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, GP., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, PY., Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on; June 2014; p. 13-24.ieeexplore.ieee.org
34. Rajagopalan, V., Boppana, V., Dutta, S., Taylor, B., Wittig, R. Xilinx Zynq-7000 EPP–an extensible processing platform family. In. 23rd Hot Chips Symposium; 2011. p. 1352-1357.
35. Rossbach, CJ., Yu, Y., Currey, J., Martin, J-P., Fetterly, D. Dandelion: a compiler and runtime for heterogeneous systems. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles; ACM; 2013. p. 49-68.
36. Sabne, A., Sakdhnagool, P., Eigenmann, R. HeteroDoop: A MapReduce programming system for accelerator clusters. Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing; New York, NY, USA. ACM; 2015. p. 235-246.HPDC '15
37. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. Proceedings of the 8th ACM European Conference on Computer Systems; ACM; 2013. p. 351-364.
38. Segal O, Colangelo P, Nasiri N, Qian Z, Margala M. SparkCL: A unified programming framework for accelerators on heterogeneous clusters. 2015 CoRR abs/1505.01120.

39. Shan, Y., Wang, B., Yan, J., Wang, Y., Xu, N., Yang, H. FPMR: Mapreduce framework on FPGA. Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays; New York, NY, USA. ACM; 2010. p. 93-102.FPGA ’10
40. Stuart, JA., Owens, JD. Multi-GPU mapreduce on GPU clusters. Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium; Washington, DC, USA. IEEE Computer Society; 2011. p. 1068-1079.IPDPS ’11
41. Tsoi, KH., Luk, W. Axel: A heterogeneous cluster with FPGAs and GPUs. Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays; New York, NY, USA. ACM; 2010. p. 115-124.FPGA ’10
42. Vavilapalli, VK., Murthy, AC., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache Hadoop YARN: Yet another resource negotiator. Proceedings of the 4th annual Symposium on Cloud Computing; ACM; 2013. p. 5
43. Wang Z, Zhang S, He B, Zhang W. Melia: A MapReduce framework on OpenCL-based FPGAs. IEEE Transactions on Parallel and Distributed Systems PP. 2016; 99:1–1.
44. Yeung, JHC., Tsang, CC., Tsoi, KH., Kwan, BSH., Cheung, CCC., Chan, APC., Leong, PHW. Map-reduce as a programming model for custom computing machines. Field-Programmable Custom Computing Machines, 2008. FCCM ’08. 16th International Symposium on; April 2008; p. 149-159.
45. Yin, D., Li, G., Huang, K-d. Scalable MapReduce framework on FPGA. In: Andreev, S.Balandin, S., Koucheryavy, Y., editors. Lecture Notes in Computer Science. Springer; Berlin Heidelberg: 2012. p. 280-294.
46. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, MJ., Shenker, S., Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation; USENIX Association; 2012. p. 2-2.
47. Zaharia, M., Chowdhury, M., Franklin, MJ., Shenker, S., Stoica, I. Spark: Cluster computing with working sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing; 2010. p. 10-10.
48. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J. Optimizing FPGA-based accelerator design for deep convolutional neural networks. Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays; New York, NY, USA. ACM; 2015. p. 161-170.FPGA ’15

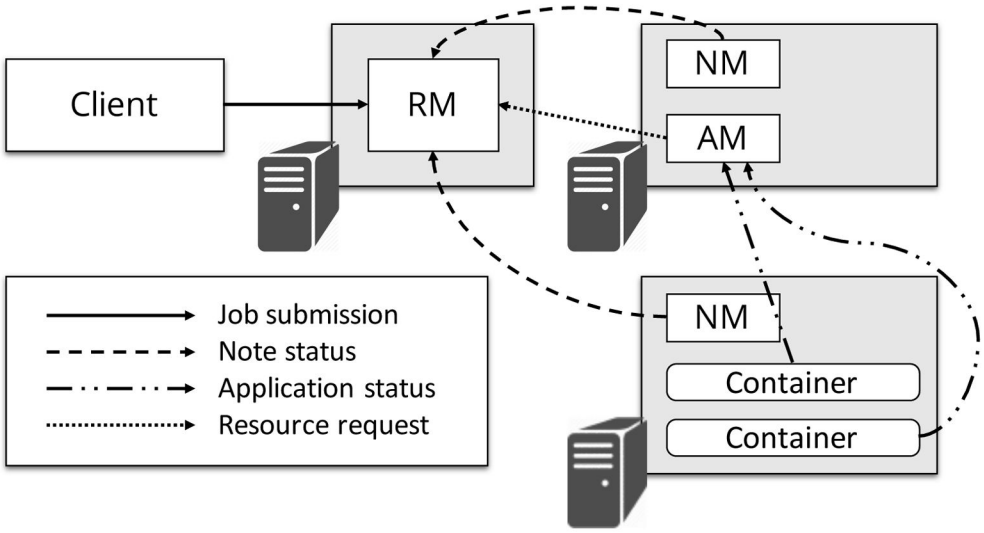
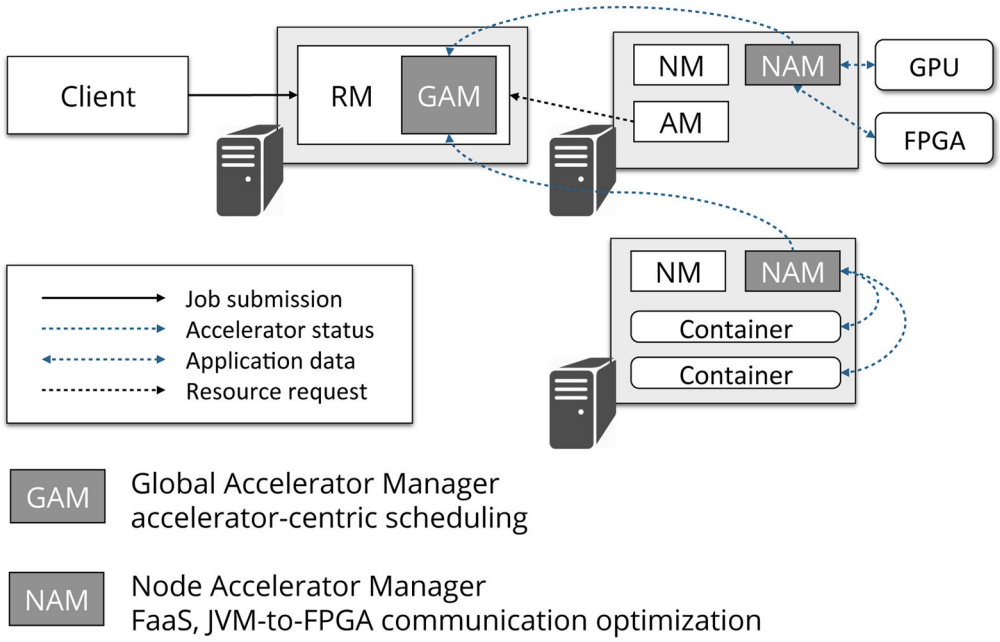


Figure 1.
Example YARN architecture showing a client submitting jobs to the global resource manager.



GAM Global Accelerator Manager
accelerator-centric scheduling

NAM Node Accelerator Manager
FaaS, JVM-to-FPGA communication optimization

Figure 2.
Overview of Blaze runtime system.

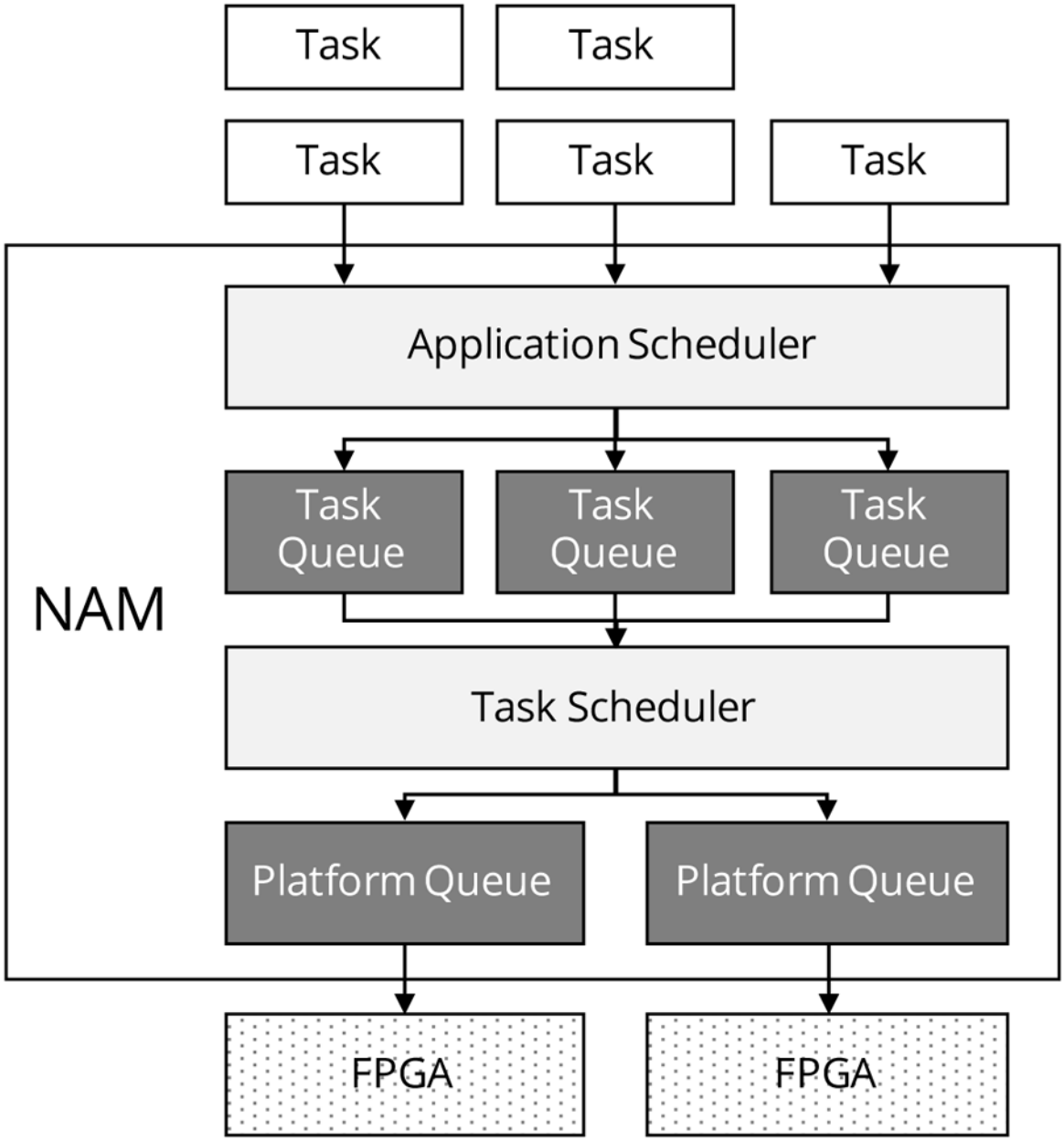
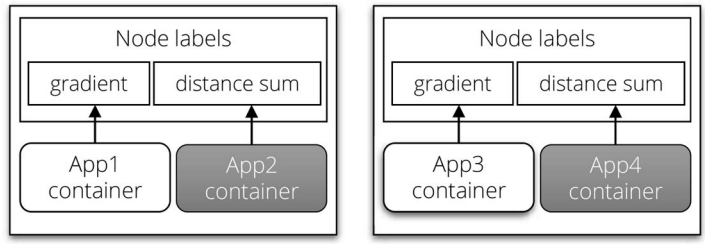
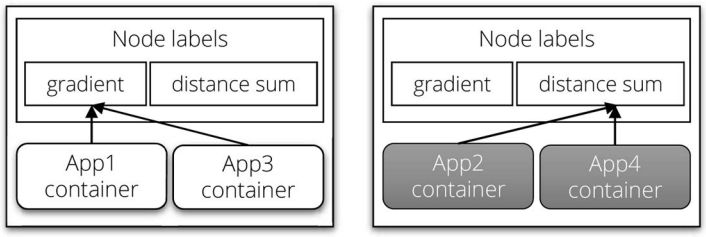


Figure 3.
Node accelerator manager design to enable FPGA accelerators as a service (FaaS).



(a) Naive allocation: applications on a node use different accelerators which leads to frequent FPGA reprogramming.



(b) Ideal allocation: applications on the node use the same accelerator and thus there is no FPGA reprogramming

Figure 4.

Different resource allocation policies. In this example, each cluster node has one FPGA platform and two accelerator implementations, “gradient” and “distance sum”. Four applications are submitted to the cluster, requesting different accelerators.

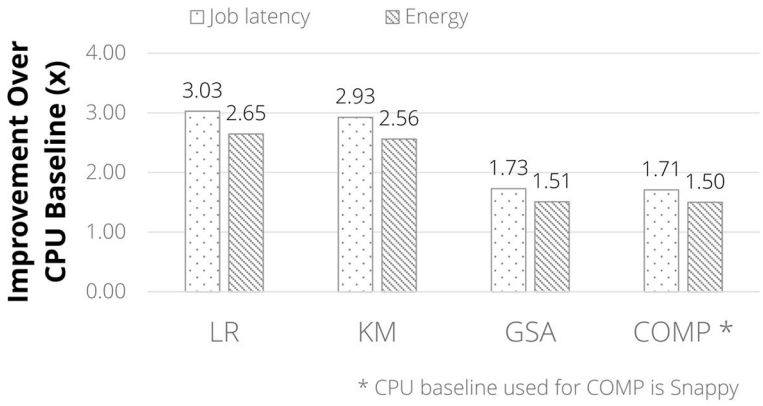
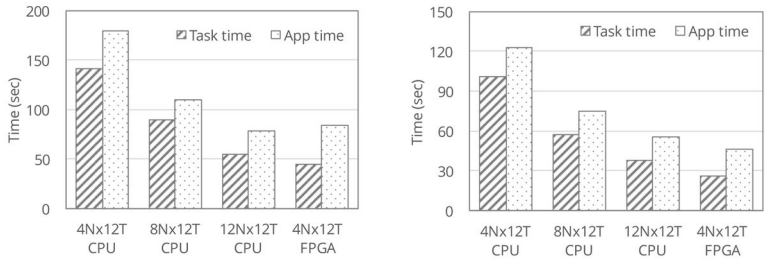


Figure 5.
Single-node system performance and energy gains for each individual application.



(a) LR (b) KM

Figure 6.

Performance of LR and KM on multiple nodes. The X-axis represents the experiment configurations. For example, ”4N×12T CPU” represents the configuration of 4 CPU-only nodes with 12 threads on each node.

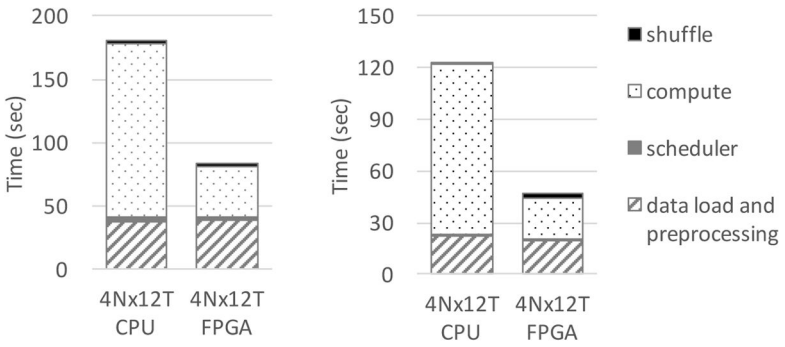


Figure 7.

Execution time breakdown for LR and KM before . and after FPGA acceleration on multiple nodes.

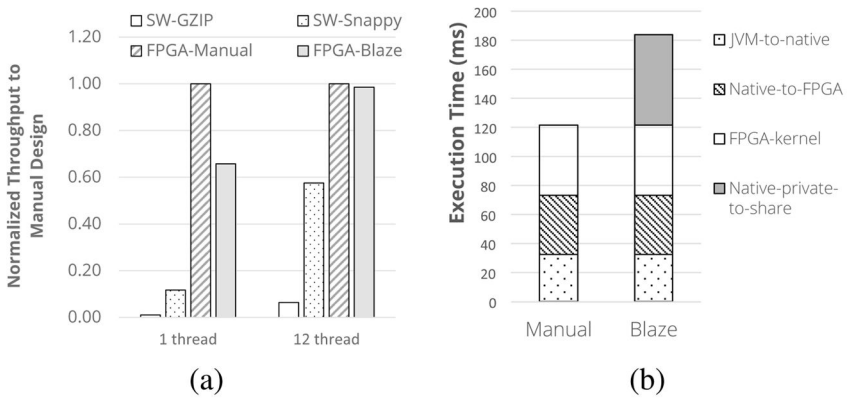


Figure 8.
Faas overhead analysis in COMP application.

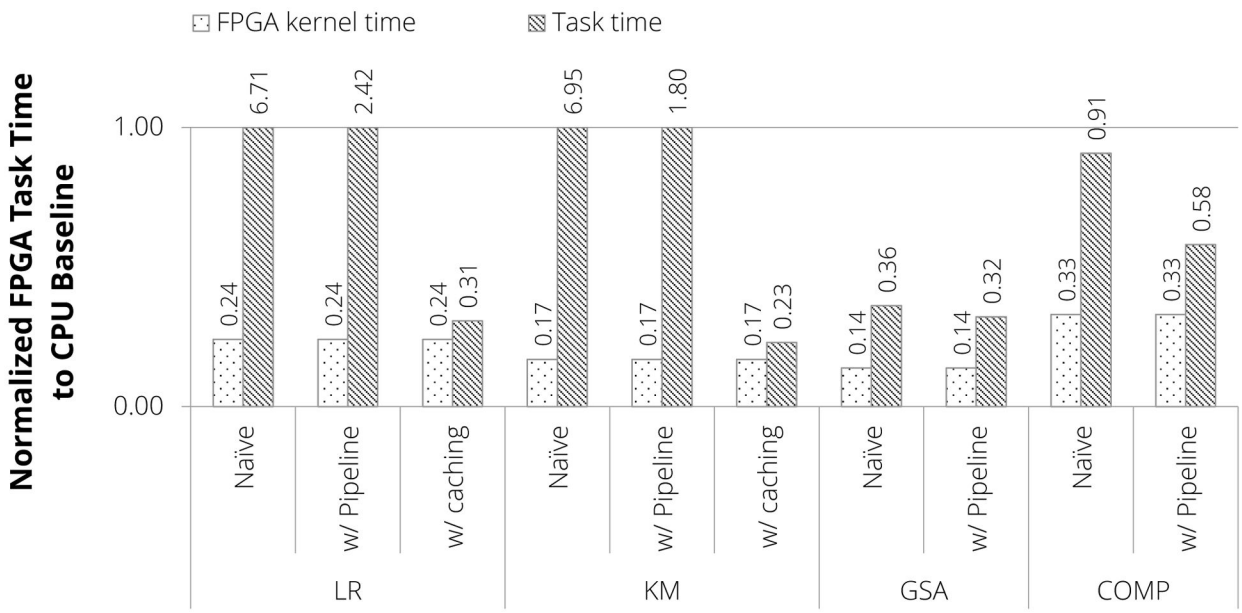


Figure 9.
Breakdown of the JVM-to-FPGA communication optimizations in FaaS.

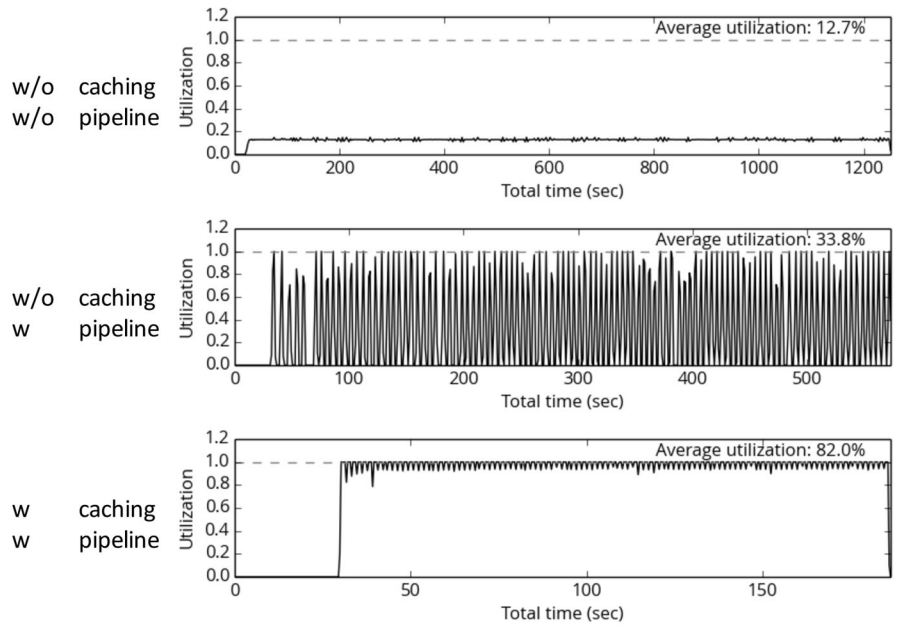
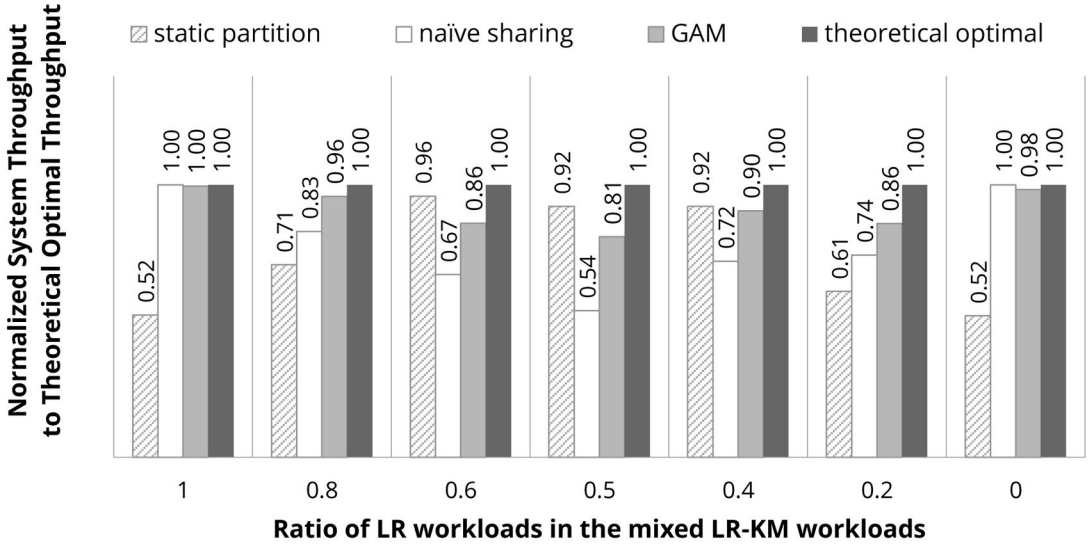
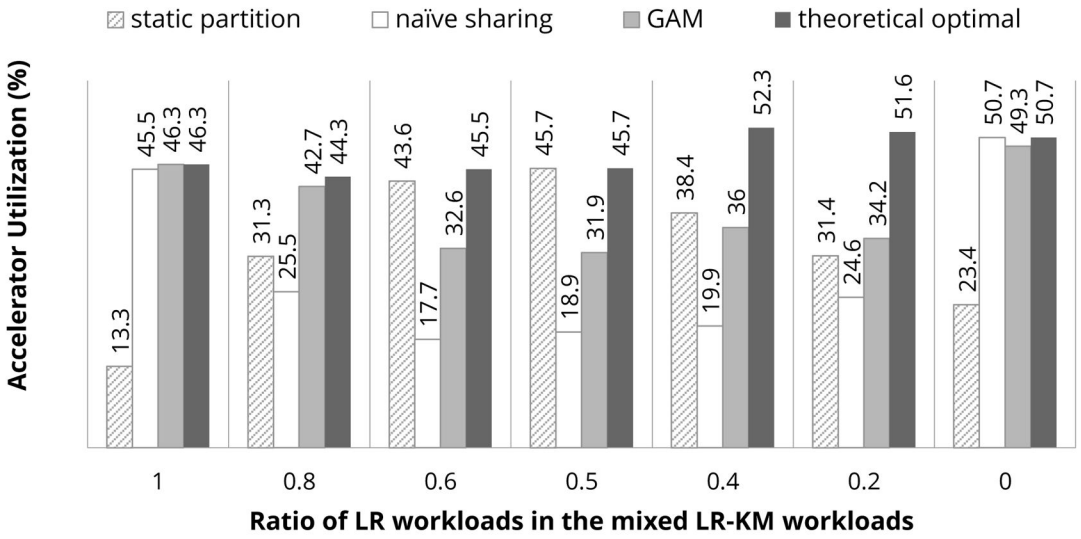


Figure 10.
Accelerator utilization results of running a single LR application on an FPGA.



(a) System throughput of different workloads. The number is normalized to the offline theoretical optimal results.



(b) Accelerator utilization of different workloads.

Figure 11.
Normalized system throughput and accelerator utilization of mixed workloads on a CPU-FPGA cluster.

Table 1

FPGA accelerator performance profile

Application	Kernel	Speedup
LR	Gradients	3.4×
KM	DistancesSum	4.3×
GSA	SmithWaterman	10×
COMP	Deflater	26.7×over Gzip 3× over Snappy

Table 2

Comparison of accelerator deployment efforts in terms of lines-of-code (LOC) changes

		App	**ACC Setup**	**Partial FaaS***
Manual	LR	26	104	325
	KM	37	107	364
	GSA	0†	227	896
	COMP	0†	70	360
		App	ACC Setup	FaaS
Blaze	LR	9	99	0
	KM	7	103	0
	GSA	0†	142	0
	COMP	0†	65	0

*Partial FaaS does not support accelerator sharing among different applications, compared to full FaaS.

†In both GSA and COMP, the accelerator kernels are wrapped as a function replacing the original software version, so no software code change is counted.