From JVM to FPGA: Bridging Abstraction Hierarchy via Optimized Deep Pipelining

Jason Cong, Peng Wei and Cody Hao Yu
{*cong, peng.wei.prc and hyu*}@*cs.ucla.edu*
University of California, Los Angeles

Abstract

FPGAs (field-programmable gate arrays) can be flexibly reconfigured to accelerate many computation kernels with orders-of-magnitude performance/watt improvement, making FPGA-based heterogeneous systems a promising approach to driving continuous performance and energy improvement in today's datacenters. However, the significant gains on computation kernels are often considerably offset by the extra data transfer overhead, resulting in considerably reduced system-wide speedup, or even slowdown. In this paper we propose a fully pipelined data transfer stack that achieves efficient JVM-FPGA communication through extensive pipelining. Also, we introduce a programming framework that automatically generates most of the pipeline code, freeing users from the bothersome details of FPGA management. Furthermore, we address the issue of multi-stage pipeline throughput optimization by formulating it into an integer linear programming problem and applying its solution for generating the optimal pipeline implementation. Experiments show that the proposed pipeline stack achieves 4.9× speedup for various computation kernels.

1 Introduction

The adoption of FPGAs in modern datacenters has attracted great attention in an attempt to drive continued performance and energy improvement. Leading datacenter operators including Microsoft and Baidu have used FPGAs to accelerate large-scale production workloads, e.g., search engines [17, 7] and neural networks [13, 14]. The Amazon Elastic Compute Cloud (EC2) also introduces the F1 instance [4] which is equipped with one or more FPGA boards. Intel, with its $16.7B acquisition of Altera, announces the Heterogeneous Architecture Research Platform (HARP) [3] where a Xeon CPU and an FPGA are connected and encapsulated in a single semiconductor package. It also predicts that 30% of datacenter servers will have FPGAs embedded by 2020 [5], indicating that FPGAs will become a major computing device in future datacenter systems.

The primary reason for the trend of FPGA adoption is that many studies from the FPGA design community have demonstrated that FPGA-based accelerators can achieve orders-of-magnitude performance/watt gains for a broad class of computation kernels [17]. However, when it comes to the integration of FPGA accelerators into the conventional software systems—especially the JVM-based big-data programming frameworks like Apache Hadoop [19] and Spark [20]—the significant improvement on the computation kernel is often considerably offset by the overhead of JVM-FPGA data communication; this results in moderate system-wide speedup or even slowdown [8, 12, 16]. This issue motivates us to develop an efficient JVM-FPGA data communication stack that will truly fulfill the orders-of-magnitude performance and energy gains on computation kernels.

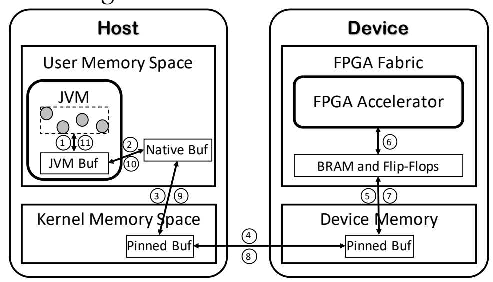


Figure 1: JVM-FPGA Data Communication Routine

Looking inside the JVM-FPGA data communication routine, we observe two impediments that result in the large overhead: 1) the overall routine is fairly complex and involves many steps of data movement, and 2) these steps are performed sequentially. Fig. 1 illustrates the entire JVM-FPGA data movement process of the conventional PCIe-based CPU-FPGA platform. In the beginning, the accelerator input data, in the form of Java objects, are packed together to be transferred out of JVM (①). As discussed in [8], this batch-processing approach is critical for improving the data communication bandwidth. The accelerator host program that directly manipulates the FPGA accelerator then receives the data from JVM (②), and initiates a PCIe-based direct memory access (DMA) to send the data to the FPGA off-chip memory (④). This DMA transfer is often coupled with a host-side memory copy (③) from the pageable space to the pinned space [9]. The data sent into the off-chip memory has to be loaded to the FPGA on-chip registers and block RAM (BRAM) (⑤), and finally be seen by the accelerator compute logic (⑥). Moreover, the generated output will go through all the above steps in the reverse direction to reach JVM (⑦-⑪), contributing the other half of the communication overhead.

The fact that these steps are performed sequentially further worsens the overall communication throughput.

Our experiments show that the throughput of the overall JVM-FPGA communication routine is only a few tens to hundreds of MB/s, or even less if the payload of each transfer is small. Meanwhile, FPGA accelerators, compared to CPUs, work at a much lower frequency and utilize deep pipelining and extensive parallelism to achieve high performance, which in turn demands high-throughput data transfer to achieve large speedup. As a consequence, the low JVM-FPGA communication throughput serves as a key issue limiting the fulfillment of the orders-of-magnitude improvement achieved by FPGA acceleration on computation kernels.

This paper proposes a high-bandwidth JVM-FPGA communication stack to address this issue. Specifically, we propose a fully pipelined JVM-FPGA communication stack that allows different jobs to be transferred and processed simultaneously, i.e., overlapping different data movement steps and the computation step. As a result, the JVM-FPGA communication throughput is greatly improved to several GB/s. Furthermore, to free users from implementing the pipeline stack that involves 1) concurrent programming in Java, C and hardware description languages, 2) FPGA runtime management, and even 3) circuit design, we propose a programming framework to automatically generate most of the pipeline code, leaving only a simple Java interface to users.

One key feature of our proposed pipeline stack is that different pipeline stages can be configured with different data transfer granularities, i.e., different payload sizes, to achieve the optimal throughput because the payload size of a data transfer stage generally determines its data transfer throughput. While it is nontrivial for programmers to manually identify the best configuration of payload sizes, we formulate the problem of pipeline throughput optimization into an integer linear programming (ILP) problem and apply its solution to pipeline code generation to achieve the optimal throughput.

While implemented for generic Java programs, the proposed pipeline stack could be particularly beneficial for cloud computing frameworks, e.g., Apache Hadoop and Spark that feature a massive degree of data-level parallelism. We discuss as future work the potential integration of the pipeline stack into these frameworks. In summary, this paper makes the following contributions:

- A JVM-FPGA communication pipeline that overlaps multiple communication and computation steps.
- A programming framework to automatically generate most of the pipeline code, freeing users from the bothersome concurrent and hardware intricacies.
- An ILP formulation of the pipeline optimization problem and automation of the optimization process.

Our experiments show that our approach achieves 4.9× speedup for a variety of computation kernels.

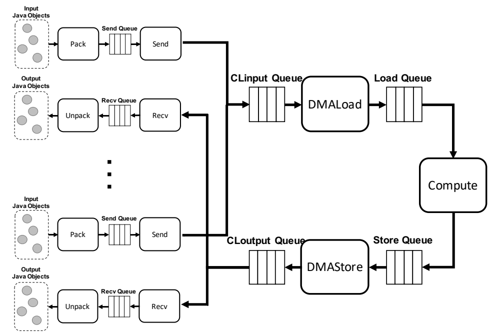


Figure 2: JVM-FPGA Pipeline Architecture

2 Pipelined Communication Stack

In this section we present our fully pipelined JVM-FPGA communication stack. Section 2.1 describes its overall architecture and major components; Section 2.2 introduces our user programming model.

2.1 System Overview

In a nutshell, the proposed approach aims to form different JVM-FPGA data movement steps and the computation step into a multistage pipeline, so the overall system performance could be determined only by the stage with the longest latency, instead of the latency of the entire JVM-FPGA routine. Fig. 2 illustrates the overall architecture of the proposed 7-stage fully pipelined JVM-FPGA communication stack. The pipeline accepts a series of Java objects that contain the input data of the FPGA accelerator, transfers the data through three pipeline stages to the FPGA fabric, performs the computation, and finally transfers the output data back to JVM through another three pipeline stages. Each stage corresponds to one or two data movement steps illustrated in Fig. 1. Every two adjacent stages are glued by a concurrent queue structure which may be implemented as software lock-free queues or hardware FIFO channels. Since the last three stages are symmetric to the first three stages, we only describe the detailed functionalities of the first four stages in the remainder of this section.

Pack. The pack stage performs data reorganization. It corresponds to ① in Fig. 1. Specifically, it retrieves the necessary input data from Java objects and puts them into a Java byte array—so it happens completely inside JVM. The byte array is then pushed into the *send queue*, a fixed-size, lock-free Java queue structure, and finally moved to the FPGA accelerator for computation. One objective of the pack stage is to achieve batch processing, i.e., batching the input of many jobs together to form a large payload to improve the data transfer throughput.

Send. The send stage accepts byte arrays from the head of the send queue, and sends them to the FPGA accelerator management program via socket. Since the host Java program, e.g., a Hadoop or Spark program, may have multiple threads using the FPGA accelerator simultaneously, we use our FPGA-as-a-service (FaaS) framework [8, 12] to realize such resource sharing. The accelerator manager in FaaS collects the data from different threads and pushes them into the *gather queue* that is a fixed-size, lock-free C++ queue structure storing OpenCL memory objects. These OpenCL memory objects are managed by the Xilinx SDAccel runtime environment [6], and stored in the pinned memory space to be transferred to the FPGA memory via PCIe. The entire stage corresponds to ②③ in Fig. 1.

DMALoad. The DMALoad stage accepts OpenCL memory objects from the gather queue and performs two data transfers. First, an OpenCL object is sent to the FPGA off-chip memory via the PCIe interface. Next, it is loaded streamingly from the off-chip memory to the *load queue* that resides in the FPGA on-chip block RAM (BRAM). The entire stage corresponds to ④⑤ in Fig. 1. The load queue is a hardware FIFO channel that connects the off-chip memory to the on-chip BRAM.

Compute. The compute stage performs the actual computation of the FPGA accelerator. It loads input data from the off-chip memory via the load queue, and stores output data back to the off-chip memory via the *store queue* that is symmetric to the load queue. The output data are then transferred through the *DMAStore*, *Recv* and *Unpack* stages back to JVM, which completes the JVM-FPGA routine.

In summary, the proposed JVM-FPGA communication stack pipelines the computation and the data transfers crossing a variety of layers, including JVM, host native memory space, FPGA off-chip memory space and on-chip BRAM. While significantly improving the JVM-FPGA communication efficiency, this heterogeneous pipeline is not easy to be manually implemented. The following section presents our programming model for the system to significantly simplify user efforts.

2.2 Programming Model

Our programming model only requires programmers to implement an application-specific interface for the *Pack* and *Unpack* stages. For example, the interface of the *Pack* stage outputs an iterator with a series of byte arrays, as shown in Code 1. In this example, we assume an Advanced Encryption Standard (AES) accelerator (see Section 4) with two arguments: `key` and `value`. The two arguments correspond to a user-defined class `StringWithKey` (line 1-4), where `value` is object-specific and `key` is shared by all `StringWithKey` objects. As can be seen in Code 1, the programmer only needs to implement a `PackIterator` with two methods. In particular, the `next` method (lines 13-29) returns one byte array at a time, where the first byte specifies which accelerator argument the byte array corresponds to. The *Pack* stage will invoke `UserPacker` iteratively and pack byte arrays with a certain size and push them to the send queue. Note that to avoid sending the shared data (i.e., `key`) redundantly, our interface provides a field `isFirstObject` to indicate whether the shared data have been sent out before.

Code 1: Programming Model with AES Example

```
1 class StringWithKey {
2  String key = ...;
3  String value = ...;
4 }
5 class UserPacker implements PackIterator {
6  int ptr = 0;
7  StringWithKey data;
8
9  public UserPacker(StringWithKey data) {
10   this.data = data;
11  }
12  public boolean hasNext() { return (ptr < 2); }
13  public Byte[] next() {
14   if (ptr == 0 && !this.isFirstObject)
15    return // Convert key to byte array
16   else if (ptr == 1)
17    return // Convert value to byte array
18   ptr++;
19   return null;
20 }}
```

By using our programming interface to specify how to pack/unpack Java objects and byte arrays, the remainder of the pipeline stack will be automatically generated. The remaining issue in pipeline generation is to determine the data transfer granularity, i.e., payload size, which determines the throughput of its corresponding pipeline stage. Since it is nontrivial for users to find the best payload size for each stage, we hide the payload size tuning from users and present our approach for automatically identifying the best configuration of payload sizes to maximize the pipeline throughput in the next section.

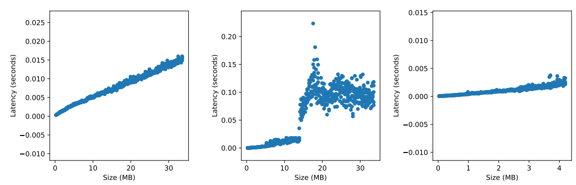
3 Pipeline Throughput Optimization

In this section we focus on the optimization of the overall pipeline throughput, i.e., the identification of the best payload sizes for all the pipeline stages. Section 3.1 first analyzes the impact of the payload size on pipeline throughput. According to the analysis, we formulate the problem to an integer linear programming (ILP) in Section 3.2 to find the best payload sizes.

3.1 Analysis of Data Transfer Throughput

In general, the latency of transferring a certain size of payload can be decoupled into two parts: 1) a constant time setup overhead, and 2) the payload movement time that is proportionate to the size of the payload. Because of the setup overhead, the data transfer throughput grows rapidly with respect to the payload size when it is small, and gradually reaches a stable value since the impact of the setup overhead is amortized. Some of the pipeline stages, e.g., the *DMALoad* stage, follow this rule very well, as is demonstrated in Fig. 3 (a). In this case, a larger payload size is always favored.

Not all the pipeline stages, however, deliver a perfect linear relation. Fig. 3 (b) shows the changes of latency



(a) DMALoad (b) Send (0∼32MB) (c) Send (0∼4MB)

Figure 3: Latency-Size Curve for Different Stages

with respect to the payload size for the *Send* stage. The payload size ranges from 0 to 32 MBs. We can see that the linear trend is not overall applicable, but still persists when the payload size is below a few megabytes, as shown in Fig. 3 (c). One possible reason is that the last-level cache is not able to hold all the intermediate data any more with the growing payload size, resulting in the sharp performance degradation in Fig. 3 (b). In this case, a larger payload size could lead to a suboptimal throughput. Moreover, the throughput optimization problem becomes even more complicated when the memory constraint is taken into consideration. Even though a larger payload often leads to a higher throughput, it also consumes more memory. Consequently, the payload size should be allocated wisely among different stages for global optimality given certain memory constraints. The following section presents our ILP-based approach to solve this problem mathematically.

3.2 Payload Size Tuning

In a nutshell, we attempt to formulate the problem of tuning the payload size of each pipeline stage into an ILP problem in which the solution can be obtained via a standard ILP solver. We present our ILP formulation for the single-core case, and will discuss the extension to the multi-core case in Section 6.

Problem Formulation: Given a computation kernel K, find a set of payload sizes $S = \{S_{pack},\ S_{send},\ \dots,\ S_{unpack}\}$ so as to maximize the overall throughput T_K. Since the throughput of a pipeline is bounded by the stage that has the minimal throughput, the overall throughput can be modeled via Eq. 1:

$$T_K = Min(T_{pack},\ T_{send},\ \dots,\ T_{unpack}) \tag{1}$$

where $T_{pack},\ \dots,\ T_{unpack}$ denote the throughputs of the seven stages, respectively. Also, we know that the throughput of a stage T_{stage} is inversely proportional to its latency L_{stage}, which can be represented as a function related to the payload size S_{stage}:

$$T_{stage} = \frac{1}{L_{stage}} = \frac{1}{f_{stage}(S_{stage})} \tag{2}$$

Therefore, to solve the problem, we need to determine each function f_{stage}.

Integer Linear Programming Formulation: To form an ILP, we model f_{stage} for each stage to a linear function while preserving the practicality and optimality.

First, the data transfer stages, i.e., *Pack/Unpack, Send/Recv* and *DMALoad/DMAStore*, have linear relations between the payload size and the latency. For the *Send/Recv* stage where the latency increases dramatically after the payload size hits a certain threshold, these large sizes can be filtered out since we can always find a better (smaller) size with a similar or higher data transfer throughput. Therefore, we are able to formulate function f_{stage} for these six stages as linear functions. Note that the *Pack/Unpack* stages are application-specific, so we profile the the application with a small dataset. The other four stages, however, are platform-specific, so we only need to profile the platform once to derive f_{stage}, which is then used for all applications running on this platform.

We then model the *Compute* stage. Depending on the time complexity of the accelerator, the computation latency may not have linear dependency to the input size. To address this issue, we profile the compute time with a set of factor-of-two input sizes, since factor-of-two data sizes generally achieve high efficiency in circuit design. Subsequently, the accelerator latency can be represented by the following linear equation:

$$L_{compute} = \sum_i p_i \times L_{S_i},\ where\ \sum_i p_i = 1,\ p_i \in \{0,1\} \tag{3}$$

where L_{S_i} denotes the latency of the i-th profiled performance point; p_i is a binary variable for each point and only one of them will be 1, i.e., only one profiled performance point with the best input size will be delivered.

Finally, we specify the memory constraint. It indicates that the overall sizes of all the queue structures cannot exceed a given memory capacity, as shown in Eq. 4:

$$\sum S_{Q_{stage}} = \sum (S_{stage} \times D_{stage}) \le S_{capacity} \tag{4}$$

where $S_{Q_{stage}}$ denotes the overall size of the queue structures for each stage and is determined by the size of each entry (S_{stage}) as well as the queue depth (D_{stage}, fixed in the proposed pipeline). Note that the software and hardware queues occupy different memory space and thus are evaluated separately.

In summary, all equations are linear manners, so the payload sizes can be determined with an ILP solver.

4 Experiments

We perform the experiments based on the PCIe-based CPU-FPGA platform that connects a Xeon CPU (E5-2420) and an Xilinx FPGA board (Alpha Data ADM-PCIE-7V3 [1]) via the PCIe interface (Gen3 x8). On top of it, we use the Xilinx SDAccel runtime environment v2017.2 [6] to drive the FPGA acceleration. On the host side, we use a set of computation kernels from the MachSuite benchmark suite [18], as listed in Table 1, to demonstrate the effectiveness of the pipeline stack on variant types of kernels. Currently, we demonstrate the effectiveness of the proposed pipeline by writing a single-thread Java program for each kernel to continuously invoke its FPGA acceleration routine, and discuss the integration with large-scale applications in Section 6.

Fig. 4 compares the execution time between the proposed pipeline stack and the conventional sequential

Table 1: Benchmark Description

Kernel	Description
AES	Advanced encryption standard.
FFT	Fast Fourier transform.
KMP	Knuth-Morris-Pratt string matching.
NW	Needleman-Wunsch sequence alignment.
STENCIL	Stencil computation.
VITERBI	Viterbi algorithm.

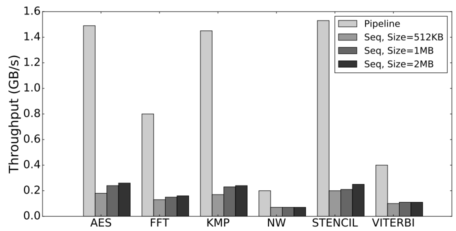


Figure 4: Throughput comparison between the pipeline and sequential JVM-FPGA communication stacks.

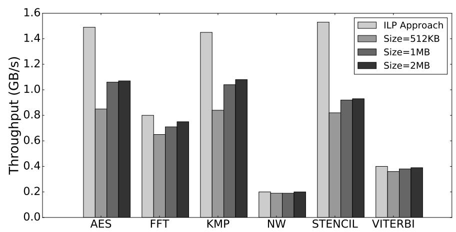


Figure 5: Throughput comparison between the proposed approach and the ad hoc solutions.

stack with 512KB, 1MB and 2MB payload sizes. We can see that the proposed pipeline stack achieves significant performance improvement on all kernels (4.9× on average), especially AES, KMP and STENCIL (5.7× to 6.1×). This is because these computation kernels are of linear time complexity, and the effective computation time after the FPGA acceleration is smaller than any of the data movement steps. The other kernels, i.e., FFT, NW and VITERBI, have super-linear time complexity, and the computation time still takes an important portion of the overall routine. Therefore, the achieved speedup (2.8× to 5.0×) is smaller, but still remarkable.

This trend is also exhibited in the throughput optimization results. Fig. 5 illustrates the performance difference between our ILP-based approach and the ones using constant-size payloads (512KB, 1MB or 2MB). We can see that the proposed approach is particularly beneficial for AES, KMP and STENCIL (34% to 65% improvement), but has moderate impact on FFT, NW and VITERBI (up to 7% improvement). This is because in the former three kernels the *Compute* stage is fully overlapped by the communication stages, allowing us to do more on throughput optimization via changing the payload sizes. On the other hand, *Compute* is still the most time-consuming stage in the latter three kernels, so having payloads with a reasonable constant size is sufficient.

5 Related Work

Recently, FPGA vendors started to provide OpenCL-based runtime environments, e.g., Xilinx SDAccel, for CPU-FPGA communication. While utilizing OpenCL's unified communication abstraction to improve programmability, these environments leave end users with the critical performance tuning issue. Also, they cannot be directly used in Java-based applications. A few previous studies aim to enable the use of FPGAs or GPUs in Java-based frameworks [11, 8, 12, 15, 10], and some of them reveal the communication issue and attempt to address it through batch processing [8, 12]. However, these studies still leave the entire routine sequentially performed. The CUDA streams technique [2] tries to hide the PCIe latency for GPUs, but does not take the whole communication routine into consideration. This motivates the proposed pipeline stack that covers all the layers of JVM-FPGA data transfer.

6 Lessons Learned and Open Discussion

This paper presents a fully pipelined communication stack to improve the JVM-FPGA data transfer efficiency, an automated programming framework for easy implementation, and an ILP-based approach for throughput optimization, achieving 4.9× speedup for a variety of computation kernels. We summarize the lessons learned and discuss some open topics as follows.

1) Embracing other CPU-FPGA platforms. The trend of adopting FPGAs in datacenters results in the release of many different CPU-FPGA platforms. Some of them, e.g., Intel HARP [3], provide a coherent, shared memory model, bringing new opportunities to the JVM-FPGA communication optimization. Also, the incorporation of FPGAs in open clouds, e.g., Amazon F1 [4], virtualizes FPGA resources to users. An extended pipeline stack that addresses the shared-memory, coherency and virtualization issues could be a challenging open topic.

2) Embracing Java-based big-data programming frameworks. There is no substantial obstacle preventing the adoption of the proposed pipeline stack from being used in Java-based domain-specific programming frameworks like Hadoop [19] and Spark [20]. In fact, their dataflow-oriented programming model could even make our user programming interface more intuitive. For example, in Spark applications we could automatically split and transfer RDDs instead of requiring users to manually implement iterators. The challenge comes from the pipeline throughput optimization, since the ILP-based formulation needs to be extended to cover various thread contention handling strategies for multithreaded programs. This remains as another open topic.

Acknowledgments

This research is partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) and contributions from Huawei, NEC and VMWare under the CDSC Industrial Partnership Program.

References

[1] Alpha Data ADM-PCIE-7V3 datasheet. http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf.

[2] Cuda toolkit documentation v8.0. http://docs.nvidia.com/cuda/.

[3] Xeon+FPGA Platform for the Data Center. https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf, 2015.

[4] Amazon EC2 F1 Instance. https://aws.amazon.com/ec2/instance-types/f1/, 2016.

[5] 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, 2016.

[6] SDAccel Development Environment. http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2016.

[7] CAULFIELD, A. M., CHUNG, E. S., PUTNAM, A., ANGEPAT, H., FOWERS, J., HASELMAN, M., HEIL, S., HUMPHREY, M., KAUR, P., KIM, J.-Y., LO, D., MASSENGILL, T., OVTCHAROV, K., PAPAMICHAEL, M., WOODS, L., LANKA, S., CHIOU, D., AND BURGER, D. A cloud-scale acceleration architecture. In *MICRO-49* (2016).

[8] CHEN, Y.-T., CONG, J., FANG, Z., LEI, J., AND WEI, P. When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (2016), USENIX Association, pp. 64–70.

[9] CHOI, Y.-K., CONG, J., FANG, Z., HAO, Y., REINMAN, G., AND WEI, P. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *DAC-53* (2016).

[10] GHASEMI, E., AND CHOW, P. Accelerating apache spark with fpgas. *Concurrency and Computation: Practice and Experience*.

[11] GROSSMAN, M., BRETERNITZ, M., AND SARKAR, V. HadoopCL2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications. *Parallel and Distributed Systems, IEEE Transactions on PP*, 99 (2015), 1–1.

[12] HUANG, M., WU, D., YU, C. H., FANG, Z., INTERLANDI, M., CONDIE, T., AND CONG, J. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), ACM, pp. 456–469.

[13] OUYANG, J., LIN, S., QI, W., WANG, Y., YU, B., AND JIANG, S. Sda: Software-defined accelerator for largescale dnn systems. In *Hot Chips* (2014).

[14] OVTCHAROV, K., RUWASE, O., KIM, J.-Y., FOWERS, J., STRAUSS, K., AND CHUNG, E. S. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *Hot Chips* (2015).

[15] PARK, J., SHARMA, H., MAHAJAN, D., KIM, J. K., OLDS, P., AND ESMAEILZADEH, H. Scale-out acceleration for machine learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), ACM, pp. 367–381.

[16] PELTENBURG, J., HESAM, A., AND AL-ARS, Z. Pushing big data into accelerators: Can the jvm saturate our hardware? In *International Conference on High Performance Computing* (2017), Springer, pp. 220–236.

[17] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., ET AL. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA-41* (2014).

[18] REAGEN, B., ADOLF, R., SHAO, Y. S., WEI, G.-Y., AND BROOKS, D. Machsuite: Benchmarks for accelerator design and customized architectures. In *IISWC* (2014).

[19] WHITE, T. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[20] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12, pp. 2–2.