INVITED: Heterogeneous Datacenters: Options and Opportunities

Jason Cong,[*] Muhuan Huang,[*†] Di Wu,[*†] Cody Hao Yu[*]
{cong, mhhuang, allwu, hyu}@cs.ucla.edu

[*]Computer Science Department
University of California, Los Angeles

[†]Falcon Computing Solutions, Inc.

ABSTRACT

In this paper we present our ongoing study and deployment efforts for enabling FPGAs in datacenters. An important focus is to provide a quantitative evaluation of a wide range of heterogeneous system designs and integration options, from low-power field-programmable SoCs to server-class computer nodes plus high-capacity FPGAs, with real system prototyping and implementation on real-life applications. In the meantime, we develop a cloud-friendly programming interface and a runtime environment for efficient accelerator deployment, scheduling and transparent resource management for integration of FPGAs for large-scale acceleration across different system integration platforms to enable "write once, execute everywhere."

1. INTRODUCTION

Computation has evolved to an unprecedented scale. Growing amounts of data are being collected from many sources such as web pages, social networks or IoT devices, while at the same time more sophisticated algorithms are being deployed in emerging applications such as customized advertisement, user behavior mining, and visual/audio recognition. Service providers such as Google, Microsoft, and Amazon are expanding their datacenter infrastructures to meet the demands. However, the semiconductor technology is reaching its physical limit of scaling [1] since the power density of a single chip is no longer able to increase. Also, energy has become the dominant limiting factor of modern datacenters. To sustain scalability, datacenters need to find and use drastic methods for reducing energy consumption.

Many recent studies have identified that emerging big-data workloads expose "scale-out" characteristics [2], for which the modern processor designs are over-provisioning [3]. Recognizing such a mismatch between the architecture and workloads, researchers have proposed to use simple, low-power CPUs as major components in datacenters [4–6]. Some large companies have also been actively exploring in this direction [7]. However, low-power CPUs may suffer from significant performance reduction for many computationally intensive tasks, which makes the overall energy-efficiency worse than high-end CPUs [8].

This paper evaluates system design options in heterogeneous datacenters with FPGA accelerators. We perform quantitative studies on a wide range of systems, including an Xeon cluster, an Xeon cluster with FPGA accelerator attached to the PCI-E bus, a low-power Atom CPU cluster, and a cluster of embedded ARM processors with on-chip FPGA accelerators. We have the following observations from the experiments:

Observation 1: By experimenting with different distributed machine learning workloads on an Intel Atom cluster and an ARM cluster, we conclude that although the small-core CPU clusters consume only 0.2× to 0.3× of the power of a server-class cluster, the performance may slow down by as much as 10× (see Section 4.3 for details). As a result, the total energy consumed by these low-power processors is still many times more than their server-class counterparts.

Observation 2: Our experiments demonstrate that poor performance of small-core CPUs can be compensated for by accelerators built on the same chip. Our prototyping cluster composed from Xilinx Zynq SoCs [9], which have both embedded ARM cores and FPGA fabrics (see Section 2 for details), demonstrated 8× to 15× speedup and energy reduction compared to Atom and ARM systems, and 2x performance gain and energy reduction compared to a regular Xeon server.

Observation 3: The FPGA accelerator is more effective for big-core systems compared to small-core systems for big-data analytic applications. One of our experiments shows that the performance of one Xeon server plus FPGA accelerator is around 2× faster than eight nodes of Zynq, even though the aggregated computing power of the FPGA fabrics on these eight nodes is around 2× more than the FPGA in the Xeon server. The main reason is that on the Zynq cluster, the sequential part of the program becomes more dominant after acceleration, and the cost of moving data to the accelerator is higher. In terms of energy efficiency, on the other hand, the difference between the two types of systems is very small.

In parallel to evaluating various options in building accelerator-rich systems, we also developed programming models and runtime management systems for the ease deploying accelerators at datacenter scale. These efforts are briefly highlighted in this paper. Conventional solutions in using accelerators in the distributed cluster-computing environment are

DAC '16, June 05 - 09, 2016, Austin, TX, USA

ISBN 978-1-4503-4236-0/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2897937.2905012

mostly based on OpenCL and MPI. However, these solutions still require programmers to have a detailed understanding of the underlying hardware architectures in order to generate high-performance accelerators. For example, MPI-based inter-node communication code needs to be rewritten when porting designs from one architecture to another.

We address these programming challenges by providing a cluster-wise accelerator programming model and runtime system, named *Blaze*, that is portable across accelerator platforms. It handles inter-node communication protocol, intra-node data movement, and accelerator task scheduling.

Blaze is designed for a recent popular cluster computing framework called *Spark* [10]. The accelerators are abstracted as subroutines for Spark tasks. These subroutines can be executed on local accelerators when they are available; when they are not, the subroutines will be executed on the CPU to guarantee application correctness. The entire process is transparent to programmers. Blaze also supports a variety of application frameworks by providing a generalized interface in C++ and Java for transparent accelerator execution.

Moreover, Blaze can save application programmers and accelerator developers from explicit communication handling between the high-level programs with low-level accelerator drivers. Most big-data frameworks like Hadoop and Spark are based on Java, so the input data needs to be preprocessed into native format and transferred to the accelerator process. This transfer may incur considerable overheads and undermine the benefit of accelerators, especially when the system is running on low-power SoCs like Zynq. Our runtime system minimizes such overheads by adopting mechanisms such as data caching and accelerator sharing.

2. SYSTEM PLATFORM OPTIONS

To evaluate the performance and energy efficiency of various accelerator-rich systems, we built several real prototype hardware systems to experiment with real-world big-data applications.

2.1 Baseline Big-Core and Small-Core Systems

For the baseline of big-core CPU systems, we built a cluster with dual-core Intel Xeon CPU servers connected with both 1G and 10G Ethernet. The cluster contains more than 20 server nodes. A snapshot of the cluster is shown in Fig. 1.

For the baseline of small-core CPU systems, we used a cluster of eight nodes of Intel Atom CPUs and a cluster of eight nodes of embedded ARM cores. The ARM cluster is the same as our prototype presented later in this section.

2.2 Big-Core with PCIE Accelerators

Similar to existing GPGPU platforms, FPGA accelerators can also be integrated into normal server nodes with PCIE slots. Taking advantage of the energy efficiency of the FPGA chips, these PCIE accelerator boards do not require an external power supply, which makes it possible to deploy FPGA accelerators into datacenters without the need to modify existing infrastructures.

In our experiments, we integrate AlphaData (AD) FPGA boards into our Xeon cluster in Fig. 1. Each FPGA board contains a Xilinx Virtex-7 XC7VX690T-2 FPGA chip with 16GB of on-board memory.

2.3 Small-Core with On-Chip Accelerators

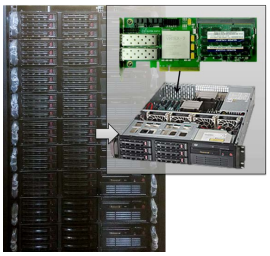
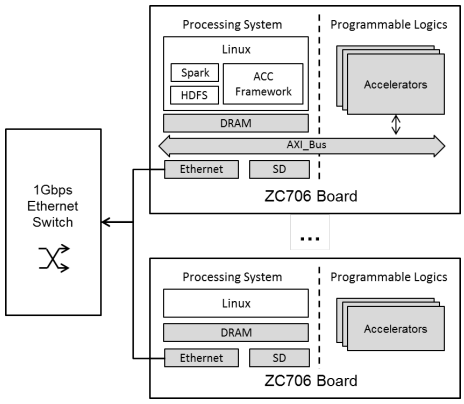


Figure 1: Experimental cluster with standard server node integrated with PCI-E based FPGA board from AlphaData

We also built a customized cluster of low-power CPU cores with on-chip FPGA accelerator. The Xilinx Zynq SoC was selected as the experimental heterogeneous SoC, which includes a processing system based on dual ARM A9 cores and a programmable FPGA logic. The accelerators are instantiated on the FPGA logic and can be reconfigured during runtime. We build a cluster of eight Zynq nodes.



(a) System overview of the prototype cluster

(b) Snapshot of the prototype cluster

Figure 2: The prototype system design

The entire hardware system is built with off-the-shelf commodity hardware. A snapshot of the system is shown in Fig. 2(b). Each node in the cluster is a Xilinx ZC706 board, which contains a Xilinx Zynq XC7Z045 chip. Each board also has 1GB of on-board DRAM and a 128GB SD card used as a hard disk. The ARM processor in the Zynq SoC shares the same DRAM controller as well as address space with the programmable fabrics. The processor can control the accelerators on the FPGA fabrics using two system buses. The memory is shared through four high-performance mem-

ory buses (HPs) and one coherent memory bus (ACP). All the boards are connected to a Gigabit Ethernet switch. The hardware layout of the Zynq boards and their connection is shown in Fig. 2(a) in the bottom box for the ZC706 board.

The software setup and accelerator integration method are shown in the upper box in Fig. 2(a). A lightweight Linux system is running on the ARM processors of each Zynq board, which provides drivers for peripheral devices such as Ethernet and SD card, and also controls the on-chip FPGA fabrics. To instantiate our machine learning accelerators on the FPGA, we design a driver module to configure the control registers of the accelerators as memory-mapped IOs, and use DMA buffers to facilitate data transfers between the host system and the accelerators. The machine learning accelerators are synthesized as FPGA configuration bitstreams and can be programmed on the FPGA at runtime.

2.4 System Profile Summary

With our prototype clusters, we can evaluate heterogeneous accelerator-rich systems on different scales. A summary of the specifications and configurations of CPU and FPGA architectures that are used in our prototype platforms are presented in Table 1 and Table 2, respectively.

Table 1: Prototype System Specifications

Item	Model	Frequency	Technology
Xeon	E5-2620v3	2.4GHz	22nm
Atom	D2500	1.86GHz	32nm
Zynq-ARM	Cortex A9	800MHz	28nm
Zynq-FPGA	XC7Z045	200MHz	28nm
AD-FPGA	XC7VX690T-2	200MHz	28nm

Table 2: Experimental Platform Configurations

Item	Configuration	Avg. Power
Big-Core	Xeon	175W
Small-Core	Atom	30W
Smaller-Core	Zynq ARM	10W
Smaller-Core+FPGA	Zynq	10W
Big-Core+FPGA	Xeon + AD-FPGA	200W

3. PROGRAMMING MODEL AND RUNTIME SYSTEM

In this section we discuss options and opportunities in programming models and runtime systems for accelerator-rich systems. Particularly, we present our software support for benchmarking big-data applications on different hardware systems.

3.1 Conventional Approaches

We found that conventional programming models used for programming accelerators in a distributed environment, such as OpenCL and MPIs, suffer from several disadvantages. First, MPI is not a friendly programming interface for heterogeneous computing. It requires excessive low-level programming efforts and is not portable across platforms; deploying the MPI framework to an SoC involves detailed platform-dependent settings and complex communication protocols between the MPI process and the FPGA accelerator.

Second, although the OpenCL framework provides a somewhat unified programming model and a runtime system for heterogeneous compute resources including multi-core CPUs, GPUs, and FPGAs on a single server node, it is still hard to adopt it in heterogeneous datacenters for the following reasons:

Portability: Although an OpenCL kernel can be compiled at runtime and runs on different platforms, the performance of the same implementation is not portable. In fact, we observed that an OpenCL program that is originally designed for GPU architecture may suffer from more than 35x slowdown against a single-thread CPU when running on the FPGA. It means that different accelerator designs need to be used on different types of accelerator platforms. This adds to the complexity of application design based on an openCL framework and can be challenging to handle at very large scales.

In addition, OpenCL does not provide good support for accelerators sharing and virtualization, and ensuring task completion when required accelerators are not available. As a result, programmers have to be responsive to these possible exceptions that cause failed executions.

Scalability: The openCL runtime system is designed for single-node execution, which means that application programmers still need to rely on inter-node communication models such as MPI in datacenters to deploy distributed applications.

Finally, from a resource management point of view, increasing accelerator utilization becomes key to system performance and throughput. However, accelerators are specialized hardware and are non-preemptive. Thus, they cannot be easily shared among multiple processes. Handling accelerator sharing among multiple cluster tenants remains an unsolved problem.

To address the above issues concerning the use of OpenCL and MPIs, we developed a new accelerator runtime system, called Blaze. Blaze is platform-independent and is portable across different systems. Together with Spark, this runtime system can scale out to a cluster computing environment.

3.2 Review: Apache Spark

Apache Spark [10] is a fast and general large-scale data processing framework. The key innovation of Spark is resilient distributed datasets (RDD) [11], which is a data collection abstraction that allows for in-memory caching of data blocks to reduce the I/O and communication overhead of large-scale data processing. In addition, Spark adapts a DAG-based scheduling scheme to manipulate RDDs efficiently. Each Spark task is constructed with a series of *transformations* on RDDs and one *action*. A transformation task generates a new RDD to represent the processed data, while an action task generates a set of results from the input RDD. Note that instead of executing each transformation as a task, Spark groups a lineage of transformations as a single task and executes the whole lineage when an action is invoked, which leads to more efficient task scheduling. Spark has demonstrated significant speedup over Hadoop MapReduce [12], especially for iterative algorithms. Currently, Spark is one of today's most popular data processing frameworks, and it is maintained by over 400 developers from more than 100 companies.

Since Spark attenuates the I/O bottleneck in big-data computation by caching RDDs in memory, the potential

benefits to accelerators are more prominent. This is the main reason that Spark is selected as a one accelerator-aware runtime system.

3.3 Runtime System Overview

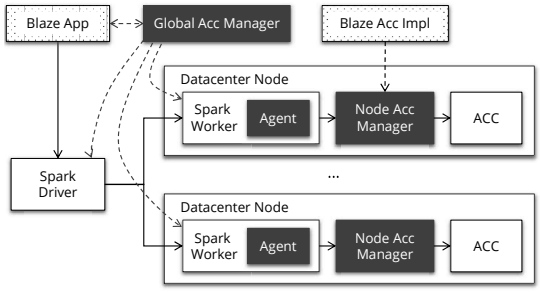


Figure 3: Overview of the Blaze runtime system

An overview of the proposed *Blaze* runtime system design is illustrated in Fig. 3. The system has three major components: a global accelerator manager, a Spark agent and a node accelerator manager. The global accelerator manager oversees the accelerator status in the cluster and decides where to launch Spark driver and the Spark workers. The Spark agent is created as an extension to the Spark RDD, which includes a communication layer that makes accelerator requests, prepares input data for accelerators and collects results. The node accelerator manager runs on each datacenter node and schedules accelerator execution based on the requests from the Spark agent. In the runtime system, cluster-level task execution is effectively managed by the Spark framework, leveraging the efficient scheduling provided by that framework. Spark tasks only send requests for accelerators within their local node to avoid inter-node communication.

3.4 Application Interface

The accelerator-aware programming model for applications is designed as an extension of Spark, with the goal of providing a clean abstraction of accelerators with a minimum of required code changes .

As mentioned in Section 3.2, a Spark program is composed from a series of transformations and actions on RDDs. We provide a set of APIs for users to extend original Spark RDDs with accelerator capability. In the APIs, an accelerator ID is provided by the user to indicate the desired accelerator functionality. This provides the runtime system capabilities to schedule tasks across different types of accelerator platforms across the datacenter, as long as the accelerators have the same interface and are marked with matching IDs. With this programming model, users can take advantage of accelerators in datacenters without knowing any implementation or platform details.

On the other hand, having a user identifying the accelerator ID can be non-trivial in practice. Alternative approaches include using higher-level APIs that expose RDD transformation as libraries, or using pattern matching to let the runtime system locate the desired accelerator automatically. We leave such possibilities for future exploration.

Finally, the programming model provides an interface for application developers to write CPU execution code that can realize the same functionality as the accelerators. In the case that accelerators are not readily available on the datacenter node due to lack of deployment, insufficient resources, or network congestion, etc., the computation tasks fall back to JVM and are executed on the CPU.

3.5 Accelerator Implementation Interface

Our programming model for accelerator design is based on a data-flow graph; each accelerator task takes a number of input data items and produces one or more output data items. Blaze exposes a few APIs for accelerator developers to describe the task input and output. Internally these APIs handle data communication between Spark programs and the accelerators. The APIs are compatible with C++ and OpenCL. With these APIs, the OpenCL platform and the data object will be managed by the runtime system instead of the developer, reducing much of the boilerplate code in the original OpenCL program. Furthermore, based on the user-provided accelerator specification, the accelerator kernel can be generated automatically from the Merlin compiler [13] by analyzing the accelerator interface.

3.6 Scheduling Optimization

As the cluster is shared by various applications which may request for different accelerators, each FPGA may be used to run different accelerators, in which case runtime FPGA reprogramming is needed. Bitstream reprogramming typically takes 1 to 2 seconds and such overhead is prominent since accelerator tasks are typically at microseconds level. To address this issue, our global accelerator manager takes reprogramming overhead into account and adopts a delay-scheduling based approach to place applications with different accelerator needs into different set of nodes.

4. EVALUATION RESULTS

4.1 Application Case Studies and Accelerator Designs

Today, machine learning has become the core of the big-data applications. The huge success of Internet services such as data-mining, web-search and advertisement drives the rapid development of machine learning algorithms. In our evaluation, we select two widely used machine learning algorithms in the experiments: logistic regression (LR) and K-Means clustering (KM).

Logistic Regression (LR): Logistic regression [14] is a combination of a linear model and a logistic function, which regulates the output between 0 and 1. The baseline of LR in our experiments is the training application implemented by Spark MLlib, with the LBFGS algorithm.

K-Means clustering (KM): K-Means clustering is an iterative algorithm which classifies data points (features) into several clusters. The baseline KM implementation in our experiments is also from Spark MLlib. The compute kernel selected is the local sum of center distances calculation. The datasets used in K-Means are the same as LR.

Both of these applications are iterative, so that they can take advantage of the data caching capability provided by the Blaze runtime system to mitigate the data transfer overheads between the host Spark program and FPGA accelerators. The input data set of the MNIST handwritten digits [15] is selected for both LR and KM.

4.2 Accelerator Kernel Design

The LR and KM accelerator kernels used in our experiments are from the Machine Learning FPGA Acceleration Library from Falcon Computing Solutions [16]. They are written as parameterized C++ code for Xilinx Vivado HLS. For all the designs on the AlphaData (AD) FPGA board, we use the Xilinx SDAccel which automatically generates the FPGA bitstream and exposes an interface to OpenCL. For Zynq designs, we directly use the Xilinx Vivado toolchain to get the bitstream.

Table 3: Specifications of our FPGA accelerators

App	Device	LUTs	FF	DSP	Speedup
LR	AD	54%	37%	45%	42x
LR	Zynq	58%	28%	48%	11x
KM	AD	34%	18%	20%	26x
KM	Zynq	22%	6%	18%	7x

Table 3 summarizes the resource consumption of our accelerator designs on different devices, as well as the speedup compared to single-core performance of the Xeon CPU in our cluster.

4.3 System Performance and Energy Results

4.3.1 Experiment Methodology

In the experiments discussed in this section, we measure the total application time, including the initial data load and communication. The energy consumption is calculated by measuring the average power consumption during operation using a power meter and multiplying it by the execution time, since we did not observe significant variations of system power during our experiments. All the energy consumption measurements also include a 24-port 1G Ethernet switch.

4.3.2 Big-Core vs. Big-Core + FPGA

We first present the effectiveness of FPGA accelerators in a common datacenter setup. Fig. 4 includes the comparison between a CPU-only cluster and a cluster of CPU with PCIE FPGA boards using LR, KM. For the machine learning workloads where most of the computation can be accelerated, FPGA can contribute to significant speedup with only a small amount of extra power. More specifically, the big-core plus FPGA configuration achieves 3.05× and 1.47× speedup for LR and KM respectively and reduces the overall energy consumption to 38% and 56% of the baseline respectively.

4.3.3 Small-Core + FPGA vs. Big-Core + FPGA

We then evaluate the performance and energy consumption between big-core with FPGA and small-core with FPGA with the application of LR and KM. Fig. 4 illustrats the execution time and energy consumption of running LR and KM applications on different systems. Notably, both the performance and energy efficiency of pure Atom and ARM clusters are worse than the single Xeon server, which confirms the argument in [8] that low-power cores could be less energy-efficient for computation-intensive workloads.

Several observations can be drawn from the results in Fig. 4. First, for both small-core and big-core systems, the FPGA accelerators provide significant performance and

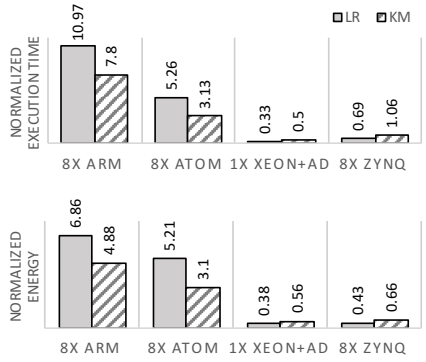


Figure 4: Execution time (above) and energy consumption (below) normalized to the results on one Xeon server.

energy-efficiency improvement—not only for kernels but also for the entire application. Second, compared to big-core systems, small-core systems benefit more from FPGA accelerators. This means that it is more crucial to provide accelerator support for future small-core-based datacenters. Finally, although the kernel performance on eight Zynq FPGAs is better than one AD FPGA, the application performance of Xeon with AD-FPGA is still 2× better than Zynq. This is because on Zynq the non-acceleratable part of the program, such as disk I/O and data copy, is much slower than Xeon. On the other hand, the difference in energy-efficiency between Xeon plus FPGA and Zynq is much smaller.

4.4 System Overheads Analysis

Here we provide more insights into the comparison between big-core plus FPGA and small-core plus FPGA by analyzing the system overheads using the example of LR. The major overheads come from the extra data copies from the Spark application to the FPGA device. The first data copy in this process is the serialization of JVM data into native format, which means consecutive data stored in memory. The second data copy is the shared memory copy between the Spark worker process and the node manager process. The last data copy is the DMA copy from node manager to the FPGA device. For AD FPGAs, this last data copy is the PCIE transfer, while for Zynq FPGAs it can be mitigated since the FPGA and ARM cores share the memory buses.

In Fig. 5 the breakdown of the execution time of one iteration of LR is presented on both the Zynq and Xeon plus AD cards. The results show that for Zynq, the proportions of the data transfer overheads are considerably larger than the Xeon case. One of the major reasons for this is the slow frequency of the memory controller. This also contributes to the reason why the energy-efficiency of small-core plus FPGA is slightly worse than big-core plus FPGA.

4.5 Summary of Different Systems

Finally, Table 4 summarizes our evaluation of different heterogeneous platforms. For our application case study of compute-intensive machine learning applications, pure small-core systems have the worst performance and consume the most energy. This means in future datacenters, it can be unrealistic to have a small-core-only configuration.

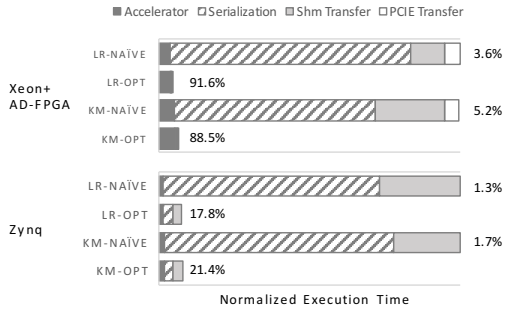


Figure 5: Execution time breakdown for Spark tasks on different platforms. The "NAIVE" implementations transfer data every iteration, while the "OPT" implementation includes the data pipelining and caching provided by the Blaze. The number on the right of each bar is the percentage of accelerator time within the total task.

Small-core plus FPGA configurations, on the other hand, have the best energy-efficiency and can also provide better performance than big-core-only configurations.

Table 4: Summary of experimental results on different platform configurations

Item	Performance	Energy-Efficiency
Big-Core + FPGA	Best	Best
Small-Core + FPGA	Better	Best
Big-Core only	Good	Good
Small-Core only	Bad	Bad

5. CONCLUSION AND FUTURE WORK

In this paper we discussed the challenges and opportunities of enabling FPGA-based accelerators as one of the building blocks in future datacenters to break the energy wall of scaling. We explored several different kinds of accelerator-rich system configurations and built prototypes to evaluate each configuration with real-world applications. We also highlighted the programmability issue for large-scale accelerator-rich systems and presented a runtime system called Blaze, which provides a high-level programming interface to Spark and automatically schedules accelerator execution without programmer interference.

Many opportunities and optimization schemes in both the hardware and software system designs can be explored. For example, the ratio between number of cores and accelerators can be optimized for different workloads. Moreover, the task scheduler of the runtime should be aware of accelerator locality in order to make intelligent scheduling for better system utilization and quality of service. Finally, as future datacenters will very likely be composed from different combinations of big-core, small-core and accelerators, resource allocation on heterogeneous systems can be a challenging problem to investigate.

6. ACKNOWLEDGMENT

This work is partially supported by the Intel Corporation with matching funds from the NSF under the Innovation Transition (InTrans) Program (CCF-1436827), and in part by C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We also want to thank Xilinx for donating the FPGA boards (ZC706 and Alphadata) used in our evaluations.

7. REFERENCES

[1] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *ISCA'11*.

[2] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, Mar. 2012.

[3] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *ISCA'10*.

[4] V. Janapa Reddi *et al.*, "Web search using mobile cores: Quantifying and mitigating the price of efficiency," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 314–325, Jun. 2010.

[5] P. Lotfi-Kamran *et al.*, "Scale-out processors," *SIGARCH Comput. Archit. News*, 2012.

[6] D. G. Andersen *et al.*, "FAWN: A fast array of wimpy nodes," in *SOSP'09*, 2009.

[7] "Baidu taps marvell for ARM storage server SoC | ee times," http://www.eetimes.com/document.asp?doc_id=1263074, accessed: 2016-3-1.

[8] L. Keys *et al.*, "The search for Energy-Efficient building blocks for the data center," in *Computer Architecture*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1 Jan. 2012, pp. 172–182.

[9] V. Rajagopalan *et al.*, "Xilinx zynq-7000 epp–an extensible processing platform family," in *23rd Hot Chips Symposium*, 2011, pp. 1352–1357.

[10] M. Zaharia *et al.*, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[11] M. Zaharia *et al.*, "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, pp. 2–2.

[12] "Apache Hadoop." [Online]. Available: hadoop.apache.org

[13] J. Cong *et al.*, "Source-to-source optimization for HLS," in *FPGAs for Software Programmers*, D. Koch *et al.*, Eds. Springer International Publishing, 2016. [Online]. Available: http://www.springer.com/us/book/9783319264066

[14] D. A. Freedman, *Statistical Models: Theory and Practice*. Cambridge University Press, 2009.

[15] Y. LeCun *et al.*, "The mnist database of handwritten digits," 1998.

[16] "Falcon Computing Solutions." [Online]. Available: http://www.falcon-computing.com