Towards FPGA-assisted Spark: An SVM Training Acceleration Case Study

Sam M.H. Ho*, Maolin Wang*, Ho-Cheung Ng*† and Hayden Kwok-Hay So*
*Department of Electrical and Electronic Engineering
The University of Hong Kong, Hong Kong
Email: {mhho, mlwang, hcng, hso}@eee.hku.hk
†Department of Computing
Imperial College London, UK
Email: h.ng16@imperial.ac.uk

***Abstract*—A system that augments the Apache Spark data processing framework with FPGA accelerators is presented as a way to model and deploy FPGA-assisted applications in large-scale clusters. In our proposed framework, FPGAs can optionally be used in place of the host CPU for Resilient distributed datasets (RDDs) transformations, allowing seamless integration between gateware and software processing. Using the case of training an Support Vector Machine (SVM) cell image classifier as a case study, we explore the feasibilities, benefits and challenges of such technique. In our experiments where data communication between CPU and FPGA is tightly controlled, a consistent speedup of up to 1.6x can be achieved for the target SVM training application as the cluster size increases. Hardware-software techniques that are crucial to achieve acceleration such as the management of data partition size are explored. We demonstrate the benefits of the proposed framework, while also illustrate the importance of careful hardware-software management to avoid excessive CPU-FPGA communication that can quickly diminish the benefits of FPGA acceleration.**

I. Introduction

Developing FPGA-accelerated applications to execute in large-scale distributed clusters while achieving reasonable speedup is a multi-faceted challenge. First, designers must parallelize their applications in preparation for distributed execution in the cluster, a process that is already challenging for many software developers. Furthermore, to achieve additional speedup with FPGA acceleration, they must also engage in the tedious hardware-software codesign and low-level hardware implementation flow. The combination of these challenges has made this process prohibitively difficult for most but a handful of highly trained specialized computer engineers.

To address these challenges, a number of attempts have been made by researchers that combine FPGA accelerator generation with programming models that are familiar to most application designers. For instances, a number of attempts have been made to develop mixed hardware-software frameworks for executing map-reduce tasks to run on hybrid FPGA-CPU clusters [1], [2], [3]. At the same time, researchers have also addressed such usability issues through integrated software-hardware generation frameworks. The LEAP project [4], for example, defines a simple computing model that is based on latency-insensitive communication channels among hardware/software modules. Similarly, the Lime language and runtime environment [5] integrates both hardware and software computation within a unified Java environment. While these frameworks have all demonstrated promising early results, in practice, the lack of integration with existing mainstream software frameworks such as Apache Hadoop remains a major usability challenge for most software-inclined application developers.

In this work, we propose a new integrated environment with the popular Apache Spark data processing framework to facilitate deployment and management of mixed hardware-software applications executing on distributed FPGA-accelerated cluster. Specifically, using the training of an SVM classifier for biological cell images [6] as a case study, we report our initial feasibility and performance trade-offs study of the proposed framework, and propose engineering practices to better manage data communication between the host CPU and FPGA accelerators to improve overall performance. In our case study, managing partition size and restricting data access to facilitate on-board data reuse played a crucial role to the resulting overall application performance. When compared to the equivalent processing on using only the host CPU cluster, our FPGA-accelerated Spark implementation of the SVM training sustained a $1.6\times$ speedup with strong scaling as the number of cluster node increases from 1 to 8.

As such, we consider the contribution of this work rests on the following aspects:

- We demonstrated the feasibility, usability and performance advantages of using FPGAs to accelerate Spark applications in a distributed cluster.
- We demonstrate engineering practices to achieve good performance when accelerating Spark applications at the level of RDD transformations.
- We demonstrate $1.6\times$ performance improvement with strong scaling by using FPGAs to accelerate an SVM cell image classifier training with minimal change to the existing software implementation.

In the next section, we will review the basic operation of the Spark application framework and SVM operation. We will then elaborate on our FPGA accelerated SVM classifier training in Section III and the target system architecture in Section IV. Experiment results will be reported in Section V and we will

978-1-5090-3707-0/16/$31.00

1: **for** $t = 1 \dots T$ **do**
2: Choose $S \subseteq N$, where $|S| = fraction \cdot n$
3: Let $L'_{w,i} = \begin{cases} -y_i x_i & \text{if } 1 - y_i(w \cdot x_i) > 0 \\ 0 & \text{otherwise} \end{cases}$
4: Set $f'_w := \frac{1}{|S|} \sum_{i \in S} L'_{w,i} + \lambda w^{(t)}$
5: Set $\gamma := \frac{step}{\sqrt{t}}$
6: Set $w^{(t+1)} := w^{(t)} - \gamma f'_w$
7: **end for**

Fig. 1. Algorithm of SVM SGD in Spark MLlib

conclude in Section VI.

II. Overview

A. Resilient distributed datasets (RDDs) in Apache Spark

The RDD [7] is a programming abstraction in Spark for a partitioned, distributed list of records. They are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them with operators. They are read-only, and can only be created from either 1) data in stable storage, e.g. Hadoop hdfs, or 2) operating on other RDDs. Such operations are called transformations, with examples like *map*, *filter*, and *join*.

Another class of operations is called actions, which return a value to the application or export data to a storage system. Examples include *count*, *collect* and *save*.

RDDs are not materialized at all times, instead their *lineage* are kept. These are the transformations an RDD was derived from, such that its partitions can be computed in the first place, re-computed when there is a fault, or when a partition is evicted from memory base on an LRU policy due to limited memory available. Thus, lost data can be recovered without requiring costly replications of RDD partitions.

B. Support Vector Machines (SVMs)

A support vector machine (SVM) is a binary classifier that finds a maximum margin separating hyperplane. The optimization problem is given as:

$$\text{minimize } \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}[1 - y_i(w \cdot x_i)]_+ \qquad (1)$$

where $[\cdot]_+$ denotes the hinge-loss:

$$[x]_+ = max(0, x)$$

w denotes the normal to the hyperplane, x_i the i-th training data and y_i its label. The above formula is also called the Primal problem for SVM, and can be think of as minimizing the hinge-loss with a l_2 regularization term. Since both the terms are convex, the Lagrangian dual of the above problem can be found and is given as:

$$\text{maxmize } \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i \cdot x_j \rangle, \qquad (2)$$
$$\text{s.t. } 0 \leq \alpha_i \leq \frac{1}{\lambda n}$$

This is called the Dual form of the SVM problem. Classical SVM solvers, like SVM-Light, Sequencial Minimal Optimization (SMO), etc., starts with this Dual problem and apply a solver on a subset of variables. However, a problem with the aforementioned classical SVM solvers is that, the runtime of these algorithms tend to scale with square of the size of the datasets, making them infeasible for large-scale datasets. For example, the run-time for SMO is $\Omega(n^2 d)$, based on emprical analysis. As such, for large-scale datasets, solvers like Pegasos [8] and LibLinear [9] were proposed. The former uses "Stochastic sub-Gradient Descent"(SGD)/ "Stochastic sub-Gradient Projection"(SGP) optimizations on the Primal problem, while the latter proposed a Dual Coordinate Descent method on the Dual form.

Pegasos guarantees the number of iterations required to obtain a solution of accuracy ϵ to be $\widetilde{O}(1/\epsilon)$, and total run-time to be $\widetilde{O}(s/\lambda\rho)$, where s is the bound on number of non-zero features in each sample, λ the SVM regularization parameter, and ρ the optimization tolerance. For LibLinear, the number of iterations needed is $O(log(1/\epsilon))$, and the run-time is $O(nd \cdot log(1/\rho))$. Therefore, in big data frameworks like Apache Spark and Apache Flink, SGD and Dual Coordinate Descent are used respectively, instead of SMO in the popular LIBSVM.

In this work, the SVM algorithm being studied is the mini-

1: $loss \leftarrow 0$
2: **for** $i \leftarrow 1 \dots n$ **do**
3: $dotp \leftarrow 0$
4: **for** $j \leftarrow 1 \dots rank$ **do**
5: $dotp \leftarrow dotp + w_i[j] \cdot x_i[j]$
6: **end for**
7: $cndt \leftarrow 1 - y_i \cdot dotp$
8: **if** $cndt > 0$ **then**
9: **for** $j \leftarrow 1 \dots rank$ **do**
10: $grdt[j] \leftarrow grdt[j] - y_i \cdot x_i[j]$
11: **end for**
12: $loss \leftarrow loss + cndt$
13: **end if**
14: **end for**

Fig. 2. Mapper process in SVM SGD

batch SGD version found in Spark's MLlib, since we are targeting big-data analytics platform for large datasets. Pseudo code of the algorithm is given in figure 1.

The $L'_{w,i}$ in line 3 is the part of a subgradient of the loss function determined by the i-th datapoint, with respect to w. f'_w in line 4 is the subgradient of the objective function. The $fraction$ variable in line 2 stands for the mini-batch fraction, which is the portion of the full training dataset to train on. When set to 1, the resulting step in each iteration is exact (sub)gradient descent, whereas when chosen very small, such that only a single point is sampled, then the algorithm is equivalent to standard SGD. The γ variable is called the learning rate, and is set to be dividing a $stepSize$ parameter by the square root of current iteration number.

When Spark executes the algorithm, the set of all training points N is represented as an RDD, as explained in Section II-B. The sampling of set S, the (sub)gradient calculation in line 3 of figure 1, and part of the summation of (sub)gradients in line 4 are parallelly executed on the cluster, by calling the "treeAggreation" function of Spark. When finished, the rest of line 4 to line 6 are calculated on a single driver node. This process repeats until the target number of iterations T is reached, or the convergence check: $\|w^{(t)} - w^{(t-1)}\| < convergenceTol * max(1, \|w^{(t)}\|)$ returns true, where $convergenceTol$ is a convergence tolerance parameter default to 0.001.

III. Accelerator Core Design

The cell images dataset we are working with consists of 256 $\times$ 256 images of 3 types of cells, 2500 for each type. This means the inputs for our SVM implementation are dense vectors of rank 65,536, which is indeed quite large if 32-bit or even 64-bit floating-point were used for each element, considering the Kintex-7 FPGA we are using offers only a total of 890 $\times$ 18k-bit $\approx$ 2MByte of on-chip BRAMs, and a double-precision vector of 65,536 $\times$ 8-Byte = 512kByte. So, the first design choice we made was to allow the input training vectors to stay as 8-bit integers for each element, since they are natively 256-level Grey-scale images. Any intermediate vector and scaler values are stored at 32-bit single-precision.

As discussed in section II-B, the computations that are parallelly scheduled over the cluster includes line 3 and part of line 4 in figure 1. These are also known as the "mappers" in map-reduce frameworks. Although not shown in the pseudo code, the function submitted to the "treeAggregate" call of Spark also returns the accumulated loss of $L = \sum_i 1 - y_i(w \cdot x_i), \forall_i \in \{(y_i, x_i) : 1 - y_i(w \cdot x_i) > 0\}$, to keep track of the actual value of the optimization target as it shrinks over iterations. As such, our implementation follows, and the logical computation in one iteration of the mapper that we aim to accelerate on FPGA is shown in the pseudo code in figure 2.

Intuitively, to implement the code in figure 2 at lowest latency, one could combine the dot product loop in line 4-6 with the one in line 9-11. This can be done by predictively doing the gradient accumulation on line 10, writing it into a temporary buffer, and only commit it if the condition $(cndt > 0)$ turns out to be true after finishing the combined loop. In hardware, commiting the temporary buffer could be implemented simply by switching the control of a ping-pong buffer. We prototyped this design and call it Ver1.

TABLE I
Utilization of implementations after synthesis compared

Designs	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Ver1	384 (43)	466 (55)	130,473 (32)	111,030 (54)
Ver2	285 (32)	464 (55)	129,677 (31)	112,289 (55)
Available	890	840	407,600	203,800

After building the Ver1, we try to integrate it into the System On-Chip, and run backend implementation flows with EDK tool. However, this design is hard to achieve timing clourse. Analyzing Ver1 we found that, the design needs significant amount of BRAMs to store the weight vector w and two gradient vectors g and g_{temp}. The three vectors alone add up to 65,536 $\times$ 4Bytes $\times$ 3 = 768kBytes, and is using 43% of on-chip BRAMs. Table I shows the resource utilizations of Ver1 and Ver2 after synthesis in Vivado HLS.

To avoid this problem, we changed to a design with longer latency, that we call Ver2. Since we cannot start the gradient accumulation until we determine the $cndt$ from the dot product, the data vector had to be stored in case later the condition $(cndt > 0)$ evaluates to true. Storing a data vector instead of a temporary gradient vector in Ver1 saved us 256kBytes - 64kBytes = 192kBytes of BRAMs, a 11% decrease. We built a fully pipelined design, which consumes all the input every cycle (II=1), and a 64kByte FIFO is used to forward the data vector to the "Axpy" module for gradient calculation, as shown in figure 3.

When the design is kickstarted, it sends command to a Xilinx DataMover core, and the initial weight vector is streamed-in from on-board DDR3-RAM through the 512-bit stream interface, followed by all the training data vectors in the RAM. Since the amount of data in a RDD partition might not fit into the DDR3-RAM, it is possible that the core is started and run multiple times within a single iteration of the algorithm.

Since each feature of our input vector is 1Byte, the 512-bit interface supplies 64 out of the 65,536 dimensions every cycle. To take advantage of this, both the buffers of the weight and gradient vectors were partitioned by 64, such that the data vectors can be processed at initialization interval (II) = 1.

At every cycle, the "PartDot" module remaps 64 dimensions of the input vector into 32-bit floating-point, multiplies them with the corresponding dimensions of the weight vector read from BRAM, and sends the results into a 64-to-1 floating-point adder tree. This outputs 1 floating-point number per cycle, which is a 64-dimension-part of the dot product between w and d (x_i in earlier formulation), to the floating-point accumulator module "FAccum". The data vector is also forwarded into a 512-bit 1024-depth FIFO, as mentioned above.

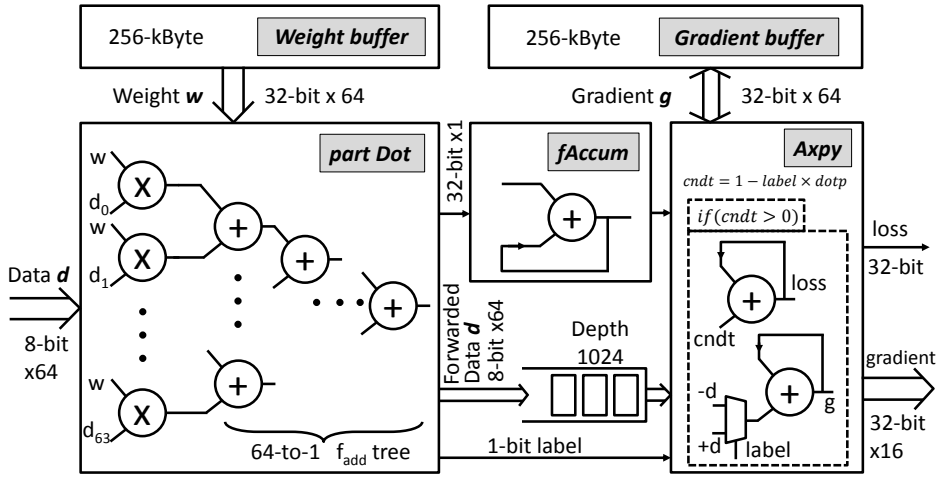

Fig. 3. Block diagram of the SVM mapper implementation

The "FAccum" module accumulates the partial dot product in 1024 cycles, and outputs the dot product to the "Axpy" module for $cndt$ calculation. The "Axpy" module starts by fetching a 1-bit "label", which is essentially the y_i in previous formulations, and then fetch the dot product and calculates $cndt$. If $cndt$ is greater than zero, it adds the (sub)gradient $(-y_i \cdot x_i)$ to the cumulative gradient in the buffer. Since we are saving calculations and wires by using 1-bit for the label, the actual implementation is a mux selecting the correspondingly-signed data, as shown in the figure 3. After processing all the training data vectors in the DDR3, the core outputs the loss and the cumulative (sub)gradient back to the DDR3.

The core was originally designed to fully exhaust the DDR3 memory controller's bandwidth, which is 512-bit at 200MHz, with an efficiency around 80 $\sim$ 90%, as estimated in section IV. However, due to the need of adding debug cores during development, we were only able to deploy a debug version of the core at 160MHz, while the core itself is believed to be capable of even higher clock rate (timing closure problems were usually due to the PCIe and DMA core at 250MHz). At 160MHz, both the DDR3 bandwidth and the SVM core could be the bottleneck, depending on the memory controller's efficiency. It should be noted that, since the SVM core is fully-pipelined, modules earlier in the pipeline ("partDot") could be processing the next data vector $d_{(i+1)}$, while modules deeper in the pipeline like ("Axpy") is still processing $d_{(i)}$, and the input bandwidth efficieny of the SVM core itself is 100% in this regard.

IV. System Architecture

Our cluster consists of 8 PC nodes, each equipped with an Intel Core i5-4570S CPU at 2.9GHz, 8GB DDR3 RAM, and a Xilinx KC705 development board featuring a xc7k325tffg900-2 FPGA with 512MB DDR3 RAM. Clustering frameworks used include Apache Hadoop, for resource management (YARN) and distributed data storage (HDFS), Apache Spark for the actual in-memory data processing, and its machine learning library (MLlib) for benchmark comparisons.

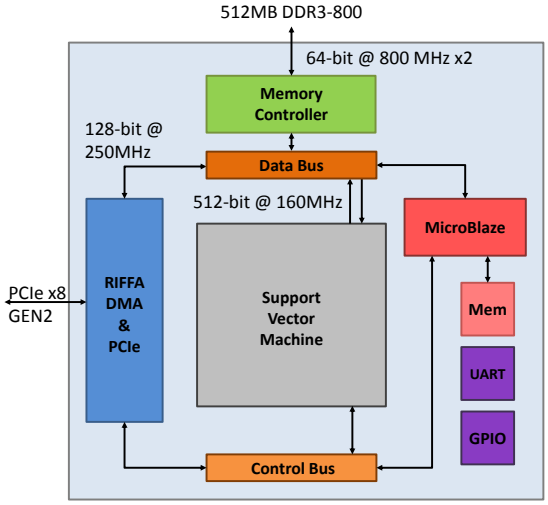

Fig. 4. Architecture of the FPGA system design

For the hardware accelerated cases, we developed our own Spark application, in which we provide a FPGA driver function to the "mapPartitions" call of Spark. The driver function iterates over the RDD partition, stores each input entries into a Java "ByteBuffer", starts a RIFFA [10] Direct Memory Access (DMA) call (which avoids copying between kernel and user space memory) to transfer it from host memory to FPGA on-board DDR3 memory, and at last transfer and return the completed result when FPGA finishes.

Here a few tricks had to be applied, and the effects will be discussed in section V. First, it is possible for Spark to schedule multiple processes to run on the same machine, but this is not supported for the hardware-accelerated cases, so we simply set the number of "vCores" in Yarn to 1 for these cases.

Second, when possible, we only transfer the training data from host to FPGA on-board memory during the first iteration of the algorithm. We call these cases "in-mem" in section V,

and is only feasible when:

$$\frac{TotalDataSize}{NumberOfNodes} <= fpgaOnboardMemorySize$$

Since it is also possible for Spark to schedule computations differently across multiple iterations, the "in-mem" trick implies that the FPGAs could well be calculating on a different partition of training data, rather than which Spark had scheduled it to. Nevertheless, in our use cases there is always at most one Spark application running on the cluster, and it uses all the assigned nodes in every iteration, so the trick does not affect the correctness of the algorithm.

Although setting a partition size (TotalSize/NumNodes) smaller than on-board memory size may not seem convincing at first, FPGA boards supporting a large amount of memories are actually available (e.g.16GB in AlphaData ADM-PCIE-7V3), and the scheme matches well with Spark's notion of in-memory processing as well, so we think this is a technique that can be useful in future systems.

The third thing to note is that, since the data on hdfs was stored in the LIBSVM format (which is in plain-text), the file size is a lot larger than the actual memory footprint when data is in binary instead. Hence, the number of partitions of a file on hdfs is larger than we expected, and so we had to apply the Spark call "coalesce", with number of nodes as argument, to reduce the partition count of the RDD before we check whether "in-mem" processing should be applied on FPGAs.

Figure 4 shows the overall architecture of our system on-chip, which was assembled using the Xilinx Vivado IP Integrator environment. The KC705 has a PCIe physical connection of x8 gen2, offering a theoratical bandwidth of 3.2GB/s. The RIFFA v2 core [10] is used for DMA between PC host memory and on-board memory. The RIFFA core runs at 250MHz, parameterized into 3 channels, and one of them was bridged onto the data bus for DDR3 access through a Xilinx DataMover core run at 240MHz. The remaining two channels offer respectively register read/write access to the control bus, and streaming access for applications like 10Gbps Ethernet, but the latter was not in-use in this context and hence not shown in the figure.

The off-chip physical interface is 64-bit at 800MHz, or 1,600M transactions per second (double data rate). This offers a 12.8GB/s maximum bandwidth. The actual achievable bandwidth depends on various factors. The overall efficiency was estimated to be 81%, when access is through a 64-bit port with burst length of 128, 10 cycles read to write overhead and 5% efficiency overhead for refresh. This gives a practical bandwidth of ∼10.37GB/s. The on-chip interface of the memory controller on the data bus is 512-bit at 200MHz.

Since this system was designed to adapt different applications, a MicroBlaze processor core running at 160MHz is also placed, and can be used to assist debugging or handle register read/writes. Table II shows the resource utilization from backend implementation of the system together with the accelerator core.

TABLE II
POST-IMPLEMENTATIONS RESOURCE UTILIZATION OF THE SYSTEM

Resource	Utilization	Available	Utilization %
LUT	124,000	203,800	60.84
LUTRAM	13,446	64,000	21.01
FF	175,229	407,600	42.99
BRAM	324	445	72.81
DSP	464	840	55.24

TABLE III
ITERATION COUNT FOR 3T3-OAC, DATA SIZE = 4800, 8 NODES

	Training Time (ms)				
Iterations	100	200	400	800	1455
CPU	21,771	41,265	76,449	141,175	243,153
FPGA	44,488	80,809	152,970	301,573	536,144
FPGA in-mem	13,586	23,936	45,187	89,420	153,668
Speedup †	1.60×	1.72×	1.69×	1.58×	1.58×
Accuracy (%)	63.24	64.81	69.12	69.85	85.29

† FPGA in-mem vs CPU

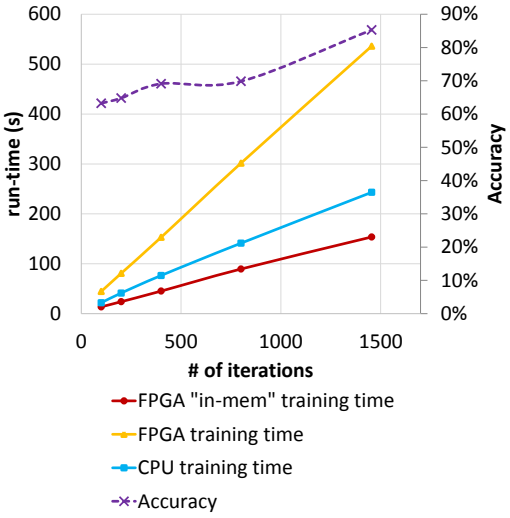

Fig. 5. Run-time(s) against iterations # for 3T3-OAC

V. BENCHMARKS

For the benchmarks, images of 3 types of cells, namely 3T3, OAC and OST, were obtained from ATOM imaging [11], and are stored in hdfs of the cluster in LIBSVM format. For the sake of simplicity, in the benchmarks here we only demonstrate 3 pairs of binary classifications.

The original dataset consists of 2,400 images for each type of cells, so for each pair of classification the dataset size is 4,800, 80% of which is used as training data, 20% as test data during run-time. As mentioned in section IV, the "in-mem" trick for FPGAs only works when partitions can fit into the on-board DRAM. In order to demonstrate the "in-mem" mode's performance, we also included a size-reduced set, where the

TABLE IV
RUN-TIME OF VARIOUS DATASETS UNDER DIFFERENT CONFIGURATIONS AGAINST NUMBER OF NODES, DATA SIZE = 2400

		Number of Nodes		
Cell Types	Iterations	2	4	8
3T3-OAC	CPU	379,141	399,016	383,630
	FPGA	1,212,213	708,175	561,779
	FPGA in-mem	194,068	216,049	244,256
	Speedup †	1.95×	1.85×	1.57×
3T3-OST	CPU	83,917	83,917	85,355
	FPGA	269,129	162,108	126,630
	FPGA in-mem	41,180	46,554	56,523
	Speedup †	2.04×	1.69×	1.51×
OAC-OST	CPU	167,898	142,636	149,764
	FPGA	496,221	284,817	229,132
	FPGA in-mem	78,066	88,580	107,572
	Speedup †	2.15×	1.61×	1.39×

† FPGA in-mem vs CPU

dataset size is 2,400, so that "in-mem" also works for 2- and 4-node configurations. These results are given in table IV.

Figure 5 shows the run-time plotted against number of iterations for the CPU, FPGA and FPGA "in-mem" cases, with 4,800 data on an 8-node config. Also overlayed onto the figure is the accuracy of the algorithm on the dataset against the number of iterations run. The figure shows that, the run-time of the SGD algorithm is almost linear with the number of iterations run, and the speedup is roughly constant with a value of $\sim$1.6× as shown in table III.

On the other hand, the speedup for FPGA "in-mem" versus CPU drops as the system scales out to more number of nodes. The values decrease from 2× to 1.49× on average.

The reason for the diminished speedup is as follows: as the number of nodes scales up, the workload scheduled onto each node decreases, but the set-up cost for FPGA accelerations, including the DMA transfers, register read/writes and function launching, etc., remains unchanged. Therefore, the ratio between computation versus setup decreases.

VI. CONCLUSION

In this paper, we have presented initial results of our proposed integrated FPGA-assisted Spark application framework. In the studied case of training an SVM cell image classifier, scalable speedup of up to 1.6× over the host CPU cluster has been demonstrated with FPGA accelerators. The speedup, however, is achievable only with very careful tuning of the Spark environment specific to the target application to ensure the best possible data reuse on the FPGA board. Although we have achieved the intended goal of acceleration, they require laborious investigation on the application data I/O run-time behavior and significant limitation on the amount of data we can efficiently process. These limitation, unfortunately counteract the benefits of the original goal to improve usability of the target FPGA accelerated cluster with common used framework like Spark. In the future, we plan to streamline this optimization process through automated accelerator generation with data I/O taken into account, and through improved communication channels between FPGAs and the storage subsystem of the cluster.

ACKNOWLEDGMENT

This work was supported in part by the Research Grants Council of Hong Kong project GRF 17245716 and the Croucher Innovation Award 2013.

REFERENCES

[1] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, "Map-reduce as a programming model for custom computing machines," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, April 2008, pp. 149–159.

[2] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: Mapreduce framework on FPGA," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723129

[3] Y. M. Choi and H. K. H. So, "Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, June 2014, pp. 9–16.

[4] K. Fleming, H. J. Yang, M. Adler, and J. Emer, "The LEAP FPGA operating system," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–8.

[5] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869469

[6] J. Xie, X. Niu, A. Lau, K. Tsia, and H. So, "Accelerated cell imaging and classification on fpgas for quantitative-phase asymmetric-detection time-stretch optical microscopy," in *Field-Programmable Technology, 2015. FPT 2015. International Conference on*, 2015, pp. 388–391.

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[8] S. Shalev-Shwartz, Y. Singer, and N. Srebro, "Pegasos: Primal estimated sub-gradient solver for SVM," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 807–814. [Online]. Available: http://doi.acm.org/10.1145/1273496.1273598

[9] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear SVM," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: ACM, 2008, pp. 408–415. [Online]. Available: http://doi.acm.org/10.1145/1390156.1390208

[10] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 22:1–22:23, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2815631

[11] T. T. Wong, A. K. Lau, K. K. Ho, M. Y. Tang, J. D. Robles, X. Wei, A. C. Chan, A. H. Tang, E. Y. Lam, K. K. Wong *et al.*, "Asymmetric-detection time-stretch optical microscopy (ATOM) for ultrafast high-contrast cellular imaging in flow," *Scientific reports*, vol. 4, 2014.