SPynq: Acceleration of Machine Learning Applications over Spark on Pynq

Christoforos Kachris
Institute of Communication and Computer Systems (ICCS/NTUA)
Athens, Greece

Elias Koromilas, Ioannis Stamelos
Department of Electrical and Computer Engineering,
National Technical University of Athens
Athens, Greece

Dimitrios Soudris
Department of Electrical and Computer Engineering,
National Technical University of Athens
Athens, Greece

***Abstract*—Spark is one of the most widely used frameworks for data analytics that offers fast development of applications like machine learning and graph computations in distributed systems. In this paper, we present SPynq: A framework for the efficient utilization of hardware accelerators over the Spark framework on heterogeneous MPSoC FPGAs, such as Zynq. Spark has been mapped to the Pynq platform and the proposed framework allows the seamlessly utilization of the programmable logic for the hardware acceleration of computational intensive Spark kernels. We have also developed the required libraries in Spark that hides the accelerator's details to minimize the design effort to utilize the accelerators.**

A cluster of 4 nodes (workers) based on the all-programmable MPSoCs has been implemented and the proposed platform is evaluated in a typical machine learning application based on logistic regression. The logistic regression kernel has been developed as an accelerator and incorporated to the Spark. The developed system is compared to a high-performance Xeon cluster that is typically used in cloud computing. The performance evaluation shows that the heterogeneous accelerator-based MpSoC can achieve up to 2.3x system speedup compared with a Xeon system (with 90% accuracy) and 20x better energy-efficiency. For embedded application, the proposed system can achieve up to 40x speedup compared to the software only implementation on low-power embedded processors and 30x lower energy consumption.

I. Introduction

Emerging applications like cloud computing, machine learning, graph computations and big data analytics require powerful systems that can process large amounts of data without consuming high power. Furthermore, these emerging applications require fast time-to-market and reduced development times. To address the large processing requirements of emerging applications, novel architectures are required in the domain of high-performance and energy-efficient processors.

Relying on Moore's law, CPU technologies have scaled in recent years through packing an increasing number of transistors on chip, leading to higher performance. However, on-chip clock frequencies were unable to follow this upward trend due to strict power-budget constraints. Thus, a few years ago a paradigm shift to multicore processors was adopted as an alternative solution for overcoming the problem. With multicore processors we could increase server performance without increasing their clock frequency. Unfortunately, this solution was also found not to scale well in the longer term. The performance gains achieved by adding more cores inside a CPU come at the cost of various, rapidly scaling complexities: inter-core communication, memory coherency and, most importantly, power consumption [1].

Therefore, the failure of Dennard's scaling, to which the shift to multicore chips is partially a response, may soon limit multicore scaling just as single-core scaling has been curtailed [2]. This issue has been identified in the literature as the *dark silicon* era in which some of the areas in the chip are kept powered down in order to comply with thermal constraints [3]. One way to address this problem is through the utilization of hardware accelerators. Hardware accelerators can be used to offload the processor, increase the total throughput and reduce the energy consumption.

In this paper we present a framework for the seamlessly utilization of hardware accelerators in heterogeneous SoCs under the Spark framework.

The main contributions of this paper are the followings:

- An efficient framework for the seamlessly utilization of hardware accelerators for Spark applications in heterogeneous FPGA-based MPSoCs
- The development of an efficient set of libraries that hide the accelerator's details to simplify the incorporation of hardware accelerators in Spark
- Mapping of the accelerated Spark to a heterogeneous 4-nodes cluster of all-programmable MPSoCs (Zynq) based on the Pynq platform
- A performance evaluation for a use-case on machine learning (logistic regression) in terms of performance and energy efficiency that shows how the proposed framework could achieve up to 2.3x speedup (for up to 90% accuracy) compared to a high-performance processor and 20x lower energy consumption.
- For embedded applications, the proposed system can achieve up to 40x system speedup compared to embedded processors and 30x better energy efficiency.

II. Related work

In the last few years, there are several efforts for the efficient deployment of hardware accelerators for cloud computing.

In [4], a detailed survey on hardware accelerator for cloud computing applications has been presented. The survey shows

both the programming framework that have been developed for the efficient utilization of hardware accelerators and the accelerators that have been developed for several applications like machine learning, graph computation applications and databases.

IBM has announced in 2016, the availability of SuperVessel cloud, a development framework for the OpenPOWER Foundation. SuperVessel has been developed by IBM Systems Labs and IBM Research based in Beijing. The goal of the SuperVessel cloud is to deliver a virtual environment for the development, testing and piloting of applications. The SuperVessel cloud framework takes advantage of IBM POWER 8 processors. Developers have access to Xilinx FPGA accelerators which use IBMs Coherent Accelerator Processor Interface (CAPI). Using CAPI an FPGA is able to appear to the POWER 8 processor as if it were part of the processor.

Xilinx has also announced in late 2016 a new framework called *Reconfigurable Acceleration Stack*. This stack is aimed at hyper scale data center that need to deploy FPGA accelerator. The FPGA boards can be hosted in typical servers and are utilized based on application specific libraries and framework integration for the five key workloads. These include machine learning inference, SQL query and data analytics, video transcoding, storage compression, and network acceleration [5]. According to Xilinx, the acceleration stack based on the FPGAs can deliver up to 20x acceleration over traditional CPUs with a flexible, reprogrammable platform for rapidly evolving workloads and algorithms.

In [6], a novel approach for integrating virtualized FPGA-based hardware resources into cloud computing systems with minimal overhead. The proposed framework allows cloud users to load and utilize hardware accelerators across multiple FPGAs using the same methods as the utilization of Virtual Machines. The reconfigurable resources of the FPGA are offered to the users as a generic cloud resources through OpenStack.

In [7], an integrated framework is presented for the efficient utilization of hardware accelerators under the Spark framework. The proposed scheme is based on a cluster-wise accelerator programming model and runtime system, named *Blaze*, that is portable across accelerator platforms. Blaze has mapped to the Spark cluster programming framework. The accelerators are abstracted as subroutines for Spark tasks. These subroutines can be executed on local accelerators when they are available. Otherwise the subroutines will be executed on the CPU to guarantee application correctness.

The proposed scheme has been mapped to a cluster of 8 Xilinx Zynq boards that host two ARM processors and a reconfigurable logic block. The performance evaluation shows that the proposed system can achieve up to 1.44× speedup for the Logistic regression and almost the same throughout for the K-Means and 2.32× and 1.55× better energy efficiency respectively. It has been also mapped to typical FPGA devices connected to the host through the PCI interface. In this case, the performance evaluation shows that the proposed system can achieve up to 3.05× speedup for the Logistic regression and 1.47× speedup for the K-Means and reduces the overall energy consumption by 2.63× and 1.78× respectively.

In this paper, we present a seamlessly utilization of hardware accelerators that can be used both for embedded systems and high-performance applications that are based on the Spark framework for computational intensive applications like machine learning and graph computation. The proposed framework allows the seamlessly utilization on the hardware accelerators based on the Spark framework using the accelerators as typical python packages.

III. Spark framework

One of the typical applications that are hosted in cloud computing is data analytics. Apache Spark [8] is one of the most widely used frameworks for data analytics. Spark has been adopted widely in recent years for big data analysis by providing a fault-tolerant, scalable and easy to use in-memory abstraction. Specifically, Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD). RDD is a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way [9]. It was developed in response to limitations in the MapReduce cluster computing framework, which forces a particular linear dataflow structure on distributed programs. MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that offers restricted form of distributed shared memory. Therefore, the latency of such applications, compared to Apache Hadoop, may be reduced by several orders of magnitude.

When the user runs an *action* (like collect), a *Graph* is created and submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. The final result of a DAG scheduler is a set of stages. The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. The *Worker* then executes the tasks for the task processing [9].

Spark libraries covers 4 main categories of applications: machine learning (MLib), graph computation (GraphX), SQL query and streaming applications.

IV. Pynq: All Programmable Systems on Chips (APSoCs)

Xilinx released in 2016 the Pynq framework that allows the utilization of the heterogeneous all-programmable SoC based on Python [10]. Using the Python language and libraries, designers can exploit the benefits of programmable logic and microprocessors in Zynq to build more capable and exciting embedded systems. Programmable logic circuits are presented as hardware libraries called *overlays*. These overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. The overlay

can be accessed through an application programming interface (API)

The Pynq platform is based on the Zynq all-programmable SoC. Zynq FPGA incorporates two RISC Cortex A9 ARM cores and a programmable logic unit in a single chip [7]. Each of these cores has 32 KB Level 1 4-way set-associative instruction and data cache and they share a 512 KB Level 2 cache. The processors are clocked at 667 Mhz and they have coherent multiprocessor support.

Zynq platform has a high performance interface for the direct communication of the ARM cores with the programmable logic part. The high performance bus is based on the ARM AMBA 3.0 interconnection that has several advantages such as QoS, multiple-outstanding transactions and low-latency paths.

V. SPYNQ: A FRAMEWORK FOR SPARK EXECUTION ON PYNQ PLATFORM

On top of the Pynq framework, we have efficiently mapped the Spark framework and we have adapted it to communicate with the hardware accelerators located in the programmable logic of the Zynq system. Spark master node is hosted on a personal computer that comes with an Intel i5 x86_64 architecture processor, but also an Intel x86 or ARM system could be used. Worker nodes are hosted on PYNQ's ARM cores.

Figure 1 shows the proposed cluster architecture. More workers beyond PYNQ cores' could be used to take advantage of all the available processing resources. This heterogeneity in workers is supported by simply adding *SPARK_WORKER_TYPE* option under nodes' *spark-env.sh* configuration. When workers start each entry of their *spark-env.sh* is added as an environment variable and can be simply accessed in Python by calling $os.environ.get()$. If 'None' or '0' (zero) is returned from $os.environ.get('SPARK\_WORKER\_TYPE')$, no programmable logic is available.

In addition, Spark comes with three different cluster managers: standalone, yarn and mesos. For the specific evaluation, standalone manager is used in client mode, meaning that the driver of Spark is launched in the same process as the client that submits the application. Each worker node is configured for starting one executor instance with all cores available. Furthermore, a python API is used for each accelerator that is used for the communication with the hardware accelerator. Each Python API is communicating with the C library that serves as the hardware accelerator driver.

On the reconfigurable logic part, the hardware accelerators for the specific application are hosted. The hardware accelerators are invoked by the python API of the Spark application. Therefore, the only modification that is required is the extension of the python library with the new function calls for the communication with the hardware accelerator.

In the typical case, the Spark application invokes the Spark MLlib and this library utilizes the Breeze library (a numerical processing library for Scala). Breeze library invokes the Netlib Java framework that is a wrapper for low-level linear algebra

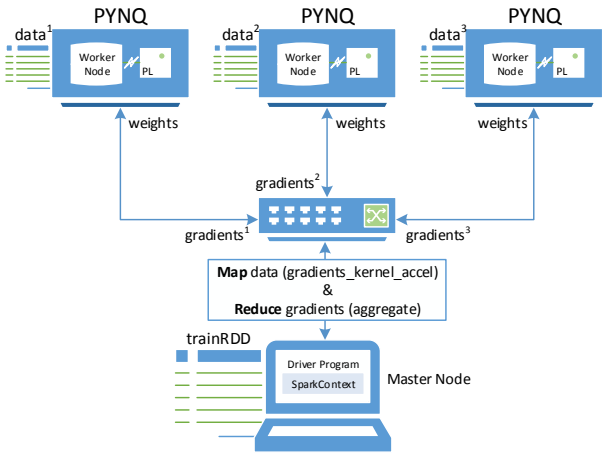

Fig. 1. Spynq architecture (LR MapReduce)

tools implemented in C or Fortran. Netlib Java is executed through the Java Virtual Machine (JVM) and the actual linear algebra tools (BLAS - Basic Linear Algebra Subprograms) are executed through the Java Native Interface (JNI).

All these layers add significant overhead to the Spark applications. Especially in applications like machine learning, where heavy computations are required, these layers add significant overhead to the kernels. Most of the clock cycles are wasted for passing through all these layers.

The utilization of hardware accelerators directly from Spark has two major advantages; firstly, the application in Spark remains as it is and the only modification that is required is the replacement of the machine learning library's function with the function that invokes the hardware accelerator. Secondly the invoking of the hardware accelerators from the python API eliminates many of the original layers thus making faster the execution of these tasks. The python API invokes the C API that serves as a hardware acceleration's library.

VI. A USE-CASE ON MACHINE LEARNING USING LOGISTIC REGRESSION

To evaluate the proposed framework, we have developed a hardware accelerator for Logistic Regression (LR) training with BGD and more specifically for the gradients kernel. The hardware accelerator has been implemented using the Xilinx Vivado High-Level Synthesis (HLS) tool. The LR application has been written in C and has been annotated with HLS *pragmas* for the efficient mapping in reconfigurable logic.

A. Algorithmic approach

Logistic Regression is used for building predictive models for many complex pattern-matching and classification problems. It is used widely in such diverse areas as bioinformatics, finance and data analytics. It is also one of the most popular machine learning techniques. It belongs to the family of classifiers known as the exponential or log-linear classifiers and is widely used to predict a binary response.

For binary classification problems, the algorithm outputs a binary logistic regression model. Given a new data point, denoted by x, where $x_0 = 1$ is the intercept term, the model makes predictions by applying the logistic function $h(z) = \frac{1}{1+e^{-z}}$, where $z = w^T x$.

By default, if $h(w^T x) > 0.5$, the outcome is positive, or negative otherwise, though unlike linear SVMs (Support Vector Machines), the raw output of the logistic regression model, $h(z)$, has a probabilistic interpretation (i.e., the probability that x is positive).

Given a training set with $numSamples$ (nS) data points and $numFeatures$ (nF) features (not counting the intercept term) $\{(x^0, y^0), (x^1, y^1), , (x^{nS-1}, y^{nS-1})\}$, where y^i is the binary label for input data x^i indicating whether it belongs to the class or not, logistic regression tries to find the parameter argument w (weights) that minimizes the following cost function:

$$J(w) = -\frac{1}{nS} \sum_{i=0}^{nS-1} \{y^i \log[h(w^T x^i)] + (1-y^i) \log[1-h(w^T x^i)]\}$$

The problem is solved using (Batch) Gradient Descent over the training set (BGD) (α is the learning rate):

1 : **procedure** $train(x, y)$
2 : initialize w with zero
3 : **while** not converged:
4 : $gradients\_kernel(x, y, w)$
5 : **for** every $j = 0, ..., nF$:
6 : $w_j -= \frac{\alpha}{nS} g_j$

7 : **procedure** $gradients\_kernel(x, y, w)$
8 : **for** every $j = 0, ..., nF$:
9 : $g_j = \sum_{i=0}^{nS-1} \{[h(w^T x^i) - y^i] x_j^i\}$

For multi-class classification problems, the algorithm compares every class with all the remaining classes (One versus Rest) and outputs a multinomial logistic regression model, which contains $numClasses$ (nC) binary logistic regression models. Given a new data point, nC models will be run, and the class with largest probability will be chosen as the predicted class.

Figure 2 depicts the high level architecture of the SPynq framework on Zynq and the block diagram of the logistic regression accelerator. The driver is used to send the parameters through the AXI interface to the hardware accelerator. In the example depicted above (Figure 2), four different channels are used for the communication between the ARM and the accelerator; two channels are used for sending the data and one channel is used for sending the weights. One more channel is used to receive the results of the accelerator (gradients). Finally, to speedup the execution time, the programmable logic hosts two copies of the kernels that can be running in parallel. Each kernel consists of four blocks that are used calculate the gradients and are pipelined to increase the overall throughput.

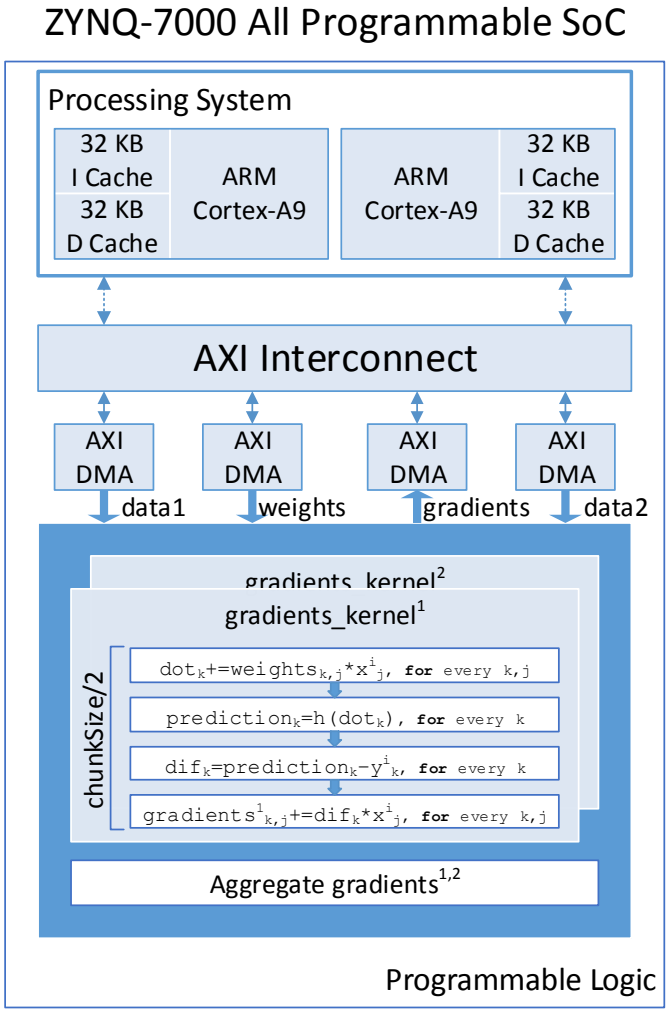

Fig. 2. Acceleration of Spark on a Zynq FPGA based on the Pynq platform. In this use-case a hardware accelerator has been developed for the Logistic Regression

B. Spark integration & Python API

In Spark $gradients\_kernel$ can be parallelized using Map-Reduce, so partial gradients are computed in each Worker, using different chunks of the training set, and then the Master aggregates them and updates w.

The Spark code for the utilization of the hardware accelerator through our accelerated machine learning library is shown in the following figure. When the Spark user wants to utilize the hardware accelerator, the only change that needs to be made is the replacement of the Spark *mllib* library with the *mllib_accel* library. Therefore, the user can speedup the execution time of the Spark application with a simple replacement of the libraries that wish to accelerate.

```
from mllib_accel.classification import LogisticRegression
from pyspark import SparkContext

sc = SparkContext(appName = "Python LR")

trainRDD = sc.textFile(train_file, numPartitions)
testRDD = sc.textFile(test_file, numPartitions)

LR = LogisticRegression(numClasses, numFeatures)
LR.train(trainRDD, chunkSize, alpha, iterations)
LR.test(testRDD)

sc.stop()
```

More specifically, in Python, a Logistic Regression object is created and various methods are supported (train, test, predict etc.). Each required action is passed in a map statement which is followed by a corresponding reduce or collect action. In example, in method train of the Logistic Regression object, gradients_kernel is mapped to all available workers and then on the workers' side SPARK_WORKER_TYPE environment variable is checked. If 'None' or '0' is returned, the whole process remains intact and the computations are executed on the CPU cores, else accelerator's specific library is called to take advantage of the programmable logic.

It is made clear that the most of the time (99,2%) is wasted on writing the train data to the allocated DMAs' buffers. Since they remain the same over the whole execution of the Logistic Regression training, we have managed and implemented a novel scheme that allow the persistent storing of the data every time the accelerator is invoked. The first approach was to create once the DMA objects and save them to a new RDD, but Spark uses by default pickle serializer that doesn't support many types of serializable items and caching this RDD would end up raising exception. One option was to create a new serializer for the specific type of returned object and add this to Spark, but doing so would end up in a very complex implementation. Contrary, we chose to implement a new function that returns the allocated buffers for the corresponding data needed for the accelerator. In that function also the train data are written into the buffers and remain there for the rest of the application execution. Every time that DMA objects are created, there is no need to create new buffers for them and fill them with the corresponding data, they just get assigned the previously created ones. Also, before destructing DMA objects, their assigned buffers are set to 'None', so that they remain intact and are not freed.

Based on the above, we have created a python API which basically consists of three calls:

- **cma** (contiguous memory allocate): This call is used for the creation of the buffers and the further allocation of contiguous memory. Also at this point overlay is downloaded and train data are written to the corresponding buffers .
- **gradients_kernel_accel**: In this call, the DMA objects are created using Xilinx's built-in modules and classes; previously allocated buffers are assigned to DMAs, current weights are written to memory and finally data are transfered to the programmable logic. Gradients are computed in return, buffers are dis-assigned from DMAs and the last ones are destructed.
- **cmf** (contiguous memory free): This call is explicitly used to free all previously allocated buffers.

It is important to note that the above demonstrated API is Spark independent and can be used in any python application.

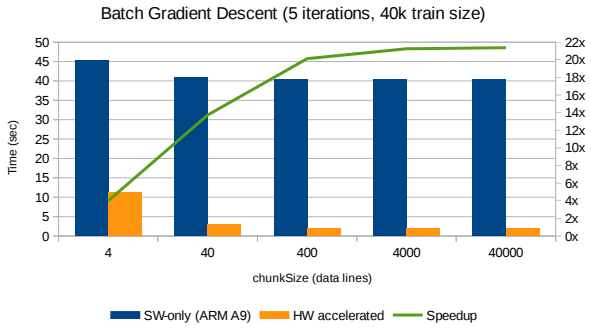

Fig. 3. Performance measurements for different chunk sizes in ZC702

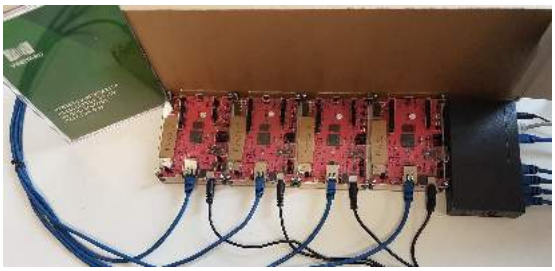

Fig. 4. Photo of the Pynq cluster

VII. Performance Evaluation

As a case study, we built a classification model with 784 features and 10 labels using 40k available training samples, for a handwritten digits recognition problem. To evaluate the performance of the system and to perform a fair comparison we built a cluster of 4 nodes with the Pynq platform (Figure 4 and we compare it with four cores using the Xeon processors. Table I shows the features of each platform. The Xeon system consists of 12 cores with 2 threads each core. The cluster made on this platform allocates 4 our of 24 threads in order to compare it with the 4 nodes of the Pynq cluster (each Pynq cluster allocates only 1 out of the 2 cores). We also compare the accelerated platform with the software only solution executed only on the ARM cores for embedded applications where only embedded processors can be used and high performance processors like Xeon cannot be supported due to power constraints.

The following paragraphs shows the performance evaluation in terms of power and execution time.

TABLE I
MAIN FEATURES OF THE EVALUATED PROCESSORS

Features	Xeon	Zynq
Vendor	Intel	ARM
Processor	E5-2658	A9
Cores (threads)	12(24)	2
Architecture	64-bit	32-bit
Instruction Set	CISC	RISC
Process	22nm	28nm
Clock Frequency	2.2 GHz	667 MHz
Level 1 cache	380 kB	32 kB
Level 2 cache	3 MB	512 kB
Level 3 cache	30 MB	-
TDP	105W	4W
Operating system	Ubuntu	Ubuntu

TABLE II
RESOURCE ALLOCATION OF THE LOGISTIC REGRESSION ACCELERATOR

Resources	Used	Total	Utilization
DSP	160	220	73%
BRAM	42	140	30%
LUT	44177	53200	83%
FF	47841	106400	45%

A. Latency and Execution time

In cases that the communication of the processor and the accelerator is often and bidirectional, this latency can be a major overhead and may diminish the speedup of the accelerator. However, in applications where the processor sends a bulk amount of data (e.g. through the AXI streaming interface), the communication overhead is overlapped by the computation time.

In the case of the Logistic Regression with BGD, the processor needs to send a large amount of data for the training of the application and therefore the communication overhead is overlapped by the computation time. In terms of resource allocation, Table II shows the utilization of the hardware resources for the Zynq FPGA SoC.

Figure 3 depicts how the performance is affected by the packet size sent to the accelerator (overlays), according to measurements performed in the Xilinx ZC702 evaluation board (C implementation) using SDSoc. As the size of the chunk (packets), that is transferred through AXI stream to the accelerator, gets smaller (such as 4 or 40 data lines), the communication overhead limits the speedup. If the size of the packets sent to the accelerator is over 400 data lines, then the communication overhead is overlapped by the computational saving in terms of execution times. The maximum kernel speedup is achieved (21×) when the packet size is over 4000 data lines (∼ 12 Mbytes). This means that by splitting each partition of the RDD into chunks between 4k and 5k lines (to make use only of simple DMAs) we can exploit our accelerator to the maximum.

Figure 5 depicts the execution time of the Logistic Regression application running on a high-performance x86_64 Intel processor (Xeon E5 2658) clocked at 2.2 GHz and on PYNQ's programmable logic for an input dataset of 40000 lines splitted in chunks of 5000 lines for various numbers of iterations (Python implementation). As it is shown, the acceleration factor is equivalent to the number of the iterations.

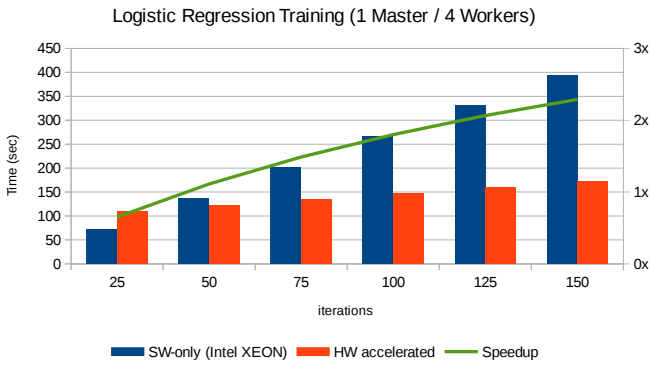

Fig. 5. Speedup versus the number of the iterations, using the proposed Python API

TABLE III
EXECUTION TIME (SEC) OF THE WORKER FUNCTIONS

Function	Xeon	ARM	Zynq (ARM+HW)
Data extraction	7.5	114	114 (ARM)
Kernel (Batch Gradient Descent) (per iteration)	2.6	45.4	0.5 (Accel)

In more detail, in Pynq the data extraction, takes about 114 sec while every iteration of the algorithm is completed in 0.45 sec since the train input data is already cached into the previously allocated buffers. On the other hand, Xeon reads and transforms the data in only 7.5 sec, but every BGD iteration takes about 2.6 sec.

So the speedup actually depends on the number of iterations that are performed. For the specific application we can achieve up to 90% accuracy with 150 iterations in which we achieve up to 2.3 system speedup compared to the Xeon cluster. However, there are applications in which much higher number of iterations are required. In that case, much higher speedup can be achieved. Table III shows the execution time of the two main functions that are executed on the worker nodes. In the Xeon platform and the ARM case, both the data extraction and the BGD are executed on the processors, while in the Zynq platform the data extraction is executed on the ARM core and the BGD function is executed on the programmable logic (accelerator). Therefore, for application where much higher number iterations are required the system speedup converges to 8.5×.

Figure 6 shows the speedup and the execution time of the accelerated platform compared to the software only solution running on the ARM processors. In this case, we can achieve up to 40× speedup compared to software only solution. This comparison is useful for applications in which

Fig. 6. Speedup versus the number of the iterations, using the proposed Python API

high-performance processors cannot be used due to power limitations (e.g. embedded systems).

B. Power and energy consumption

To evaluate the energy savings we have also measured the power consumption using the ZC702 board which hosts the same Zynq device as the Pynq platform. We measured the average power running the algorithm both in the SW-only (for the Xeon and the ARM processors), and the HW accelerated case.

Figure 7 shows the dynamic power consumption of the software only and the accelerated cases. In the first case, the LR kernel executed on the ARM cores takes around 204 seconds to complete and the total power consumption is 2.3 Watts, resulting to 469.1 Joules. In the second case, the kernel is executed in the programmable logic using the hardware accelerator. As we can see, in this case, the power consumption has been increased to 3.1 Watts. However, in that case the total execution time is 9.5 seconds thus the total energy consumption drops to 29.3 Joules. Therefore, the proposed scheme can not only speedup the execution time but also can achieve up to 16× energy efficiency.

Figure 8 shows the energy consumption comparison between the Xeon and the Zynq platform and Figure 9 shows the energy consumption comparison between the ARM and the Zynq platform. In the first case, the average power consumption of Xeon processors and the DRAMs is 103 Watt while the Zynq platform (both the MPSoC FPGA and the DRAM) is 3 Watt. In that case, we can achieve up to 20× better energy efficiency due to the lower power consumption and the lower execution time. In order to measure the energy consumption of the Xeon server, Intels Processor Counter Monitor (PCM) API is used. Among others, PCM API enables capturing the energy consumed by the CPU and DRAM memory for executing an application.

Figure 9 shows the energy consumption comparison between the ARM and the Zynq platform. In the this case the power consumption of the accelerated platform (Zynq) is slightly higher than the power consumption of the ARM-only

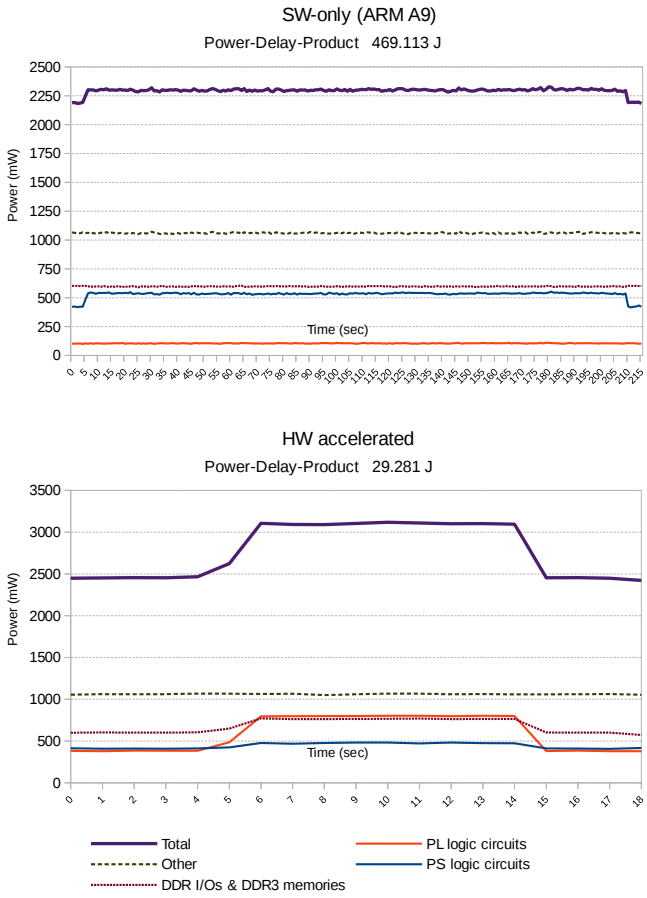

Fig. 7. Comparison of the power consumption in the SW-only (ARM) and the accelerated version

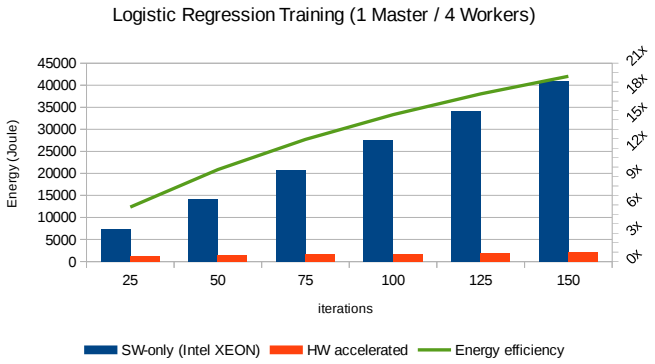

Fig. 8. Energy consumption of the Xeon and the Zynq platform based on the number of iterations

platform, but due to the significant much higher execution time of the ARM-only solution we can achieve up to 30× lower energy consumption.

Fig. 9. Energy consumption of the ARM-only and the Zynq platform based on the number of iterations

VIII. Conclusions

Hardware accelerators can improve significantly the performance and the energy efficiency of data analytic applications. However, currently data analytics frameworks like Spark do not support the seamlessly utilization of hardware accelerators. In this paper we have a present a novel scheme for the seamlessly utilization of hardware accelerators using the Spark framework that is widely used in data analytics. We have implemented a hardware accelerator for logistic regression that is connected to processors through the AXI interface and we have integrated the accelerator with the Spark framework in a cluster of all-programmable MPSoCs.

The proposed system can be used both in high performance systems to reduce the energy consumption (up to 20×) and also reduce up to 2.3× the execution time, while in embedded systems it can achieve up to 40× speedup compared to the embedded processors and up to 30× lower energy consumption for 150 iterations.

It also shows that the proposed framework can be utilized to support any kind of hardware accelerators in order to speedup the execution time of computational intensive machine learning and data analytics applications based on Spark.

IX. Acknowledgments

This project has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No 687628 - VINEYARD: Versatile Integrated Accelerator-based Heterogeneous Data Centers www.vineyard-h2020.eu. We would like to acknowledge Xilinx University Program for the kind donation of the software tools and hardware platforms.

References

[1] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[2] Christian Martin. Post-Dennard Scaling and the final Years of Moores Law. Technical report, 2014.

[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, May 2012.

[4] C. Kachris and D. Soudris. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, Aug 2016.

[5] Xilinx reconfigurable Acceleration Stack targets machine learning, data analytics and Video Streaming. Technical report, 2016.

[6] S. Byma, J.G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116, May 2014.

[7] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. Invited - heterogeneous datacenters: Options and opportunities. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 16:1–16:6, New York, NY, USA, 2016. ACM.

[8] Apache, spark, http://spark.apache.org/.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[10] Pynq: Pyhton productivity for Zynq. Technical report, 2016.