Spark acceleration on FPGAs: A use case on machine learning in Pynq

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

Christoforos Kachris
Institute of Communication and
Computer Systems (ICCS/NTUA)
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. Spark allows fast development for several applications like machine learning, graph computations, etc. In this paper, we present Spynq: A framework for the efficient deployment of data analytics on embedded systems that are based on the heterogeneous MPSoC FPGA called Pynq. The mapping of Spark on Pynq allows that fast deployment of embedded and cyber-physical systems that are used in edge and fog computing. The proposed platform is evaluated in a typical machine learning application based on logistic regression. The performance evaluation shows that the heterogeneous FPGA-based MPSoC can achieve up to 11x speedup compared to the execution time in the ARM cores and can reduce significantly the development time of embedded and cyber-physical systems on Spark applications.**

I. Introduction

Emerging applications like cloud computing, big data analytics, and IoTs 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 that are used to speedup the processing of Spark data analytics applications.

The main contributions of this paper are the followings:

- An efficient framework for the seamlessly utilization of hardware accelerators for Spark applications
- An implementation to a highly heterogeneous all-programmable MPSoC (Zynq) based on the Pynq framework
- A performance evaluation for a use-case on machine learning that shows how the proposed framework could achieve up to 11x speedup compared to the ARM processors in Zynq.

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. They 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.

978-1-5090-4386-6/17/$31.00 ©2017 IEEE

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 this paper, we present a seamlessly utilization of hardware accelerators that can be used both for embedded systems and high-performance applications like cloud, edge and fog computing. The proposed framework allows the seamlessly utilization on the hardware accelerators based on the Spark framework using the accelerators as python packages.

III. Spark framework

One of the typical applications that are hosted in cloud computing is data analytics. Apache Spark [7] 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 [8]. 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 [8].

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 fast programming of the heterogeneous all-programmable SoC [9]. 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.

Figure 1 depicts the high level architecture of the SPynq framework and the mapping on the Zynq platform. Spark is hosted on the ARM processors, on top of the Python VM that is hosted on Ubuntu OS. Both the master and the worker nodes are hosted on the ARM cores. 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 by the Spark user is the replacement of the library function calls with the python API for 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

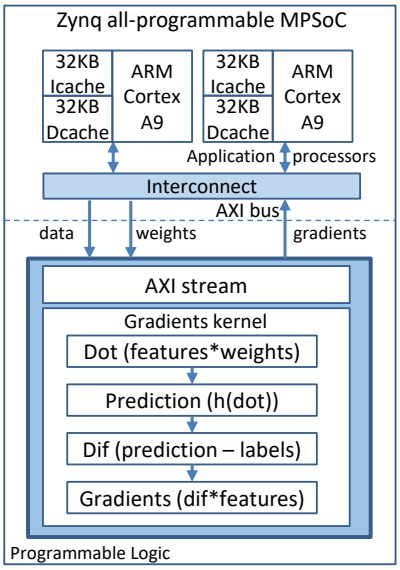

Fig. 1. Acceleration of Spark on a Zynq FPGA platform based on Pynq

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). It is interesting to note that BLAS can be written in C or even Fortran.

All these layers add a significant overhead to the spark applications. Especially in applications like machine learning, where heavy computations are required, these layers add a significant overhead for the fast execution of the tasks. 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. The driver is used to send the parameters through the AXI interface to the hardware accelerator.

The modifies functions for the execution of the function in the hardware accelerator, need to convert the RDDs to an array of variables, send the RDDs to the hardware accelerator and then convert back the results to RDDs that can be further processed by the rest of Spark program.

VI. A USE-CASE FOR MACHINE LEARNING BASED ON LOGISTIC REGRESSION

To evaluate the proposed framework, we have developed a hardware accelerator for machine learning based on logistic regression. The hardware accelerator has been developed using the High-Level Synthesis tool (Vivado HLS) from Xilinx. The logistic regression application has been written in C and has been annotated with HLS *pragmas* for the efficient mapping in reconfigurable logic.

Logistic regression measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function, which is the cumulative logistic distribution. Like other forms of regression analysis, logistic regression makes use of one or more predictor variables that may be either continuous or categorical. Unlike ordinary linear regression, however, logistic regression is used for predicting binary dependent variables rather than a continuous outcome.

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, 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, the raw output of the logistic regression model, h(z), has a probabilistic interpretation (i.e., the probability that xis positive).

Given a training set with n-data points and m-features $(x^(0,)y^0), (x^(1,)y^1), , (x^(n-1), y^(n-1))$, 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 that minimizes the following cost function:

$$J(w) = -\frac{1}{n}\sum y^i log(h(W^T x^i)) + (1 - Y^i) log(1 - h(w^T x^i)) \quad (1)$$

The problem is solved using gradient descent over the training set:

$$gradients_m = -\frac{1}{n}\sum (h(W^T x^i) - y^i) x_m \quad (2)$$

For multi-class classification problems, the algorithm will output a multinomial logistic regression model, which contains k-binary logistic regression models. Given a new data point, k-models will be run, and the class with largest probability will be chosen as the predicted class.

Figure 2 shows the original and the modified code for the utilization of the hardware accelerator in the Spark code. In the original code, the logistic regression libraries are used and the specific functions are called for the training of the application.

In the modified example, we first import the Pynq libraries that are used for the configuration of the FPGA and for the interface with the accelerators. We also import the specific API library that is used for the communication of the processor with the logistic regression hardware accelerator. The only other modifications that we need to make is the

Original code

```
import
org.apache.spark.mllib.linalg.BLAS.dot

…
Def train(
New LogisticRegressionWithBGD
(stepSize, numIterations, 0.0,
    miniBatchFraction)
      .run(input, initialWeights)
…
```

Modified code

```
import
org.apache.spark.mllib.linalg.BLAS.dot
from pynq import PL, Overlay
from pynq.board import logistic

// download bitstream
Accel_Initialization (logistic.bit)

Def train(
…
// return weights
Model=LogisticRegressionAccel.train(dataset)
…
```

Fig. 2. Original and modified code for the logistic regression example

replacement of the functions that we want to execute with the specific API for the utilization of the hardware accelerator (i.e. $accel_logistic_regression$).

The python API that is developed pass the parameters to the C API that serves as driver for the hardware accelerator. The C API is used to pass the parameters through the AXI interface to the hardware accelerator from the processors. After the hardware accelerator has finished the execution of the task, the results are again returned to the processor through the AXI bus, then the C API and finally through the Python API.

VII. Performance evaluation

As a case study, we built a classification model with 784 features and 10 labels using 40k training samples, for a handwritten digits recognition problem. The following paragraphs shows the performance evaluation in terms of latency and execution time.

A. Latency and Execution time

For the efficient utilization of hardware accelerators, low-latency communication is required between the host processor and the accelerator. To evaluate the latency of the communication between Spark and the accelerator we measured the time to send a single 32-bit word in the programmable logic. The overall time is around 300 usec including the time for the python API, the C API, the latency of the JNI, and the latency of the AXI bus. 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 send 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, 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 I shows the utilization of the hardware resources for the Zynq FPGA SoC.

TABLE I
Resource allocation of the logistic regression accelerator

Resources	Used	Total	Utilization
DSP	160	220	72%
BRAM	84	140	60%
LUT	47809	53200	90%
FF	47446	106400	45%

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 single SoC that support programmable logic. The performance evaluation shows up to 11× speedup for the logistic regression and shows that the proposed framework can be utilized to support any kind of hardware accelerators.

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

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

[8] 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.

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