Efficient hardware acceleration of recommendation engines: a use case on collaborative filtering

Konstantinos Katsantonis, Christoforos Kachris, and Dimitrios Soudris

[1] National Technical University of Athens (NTUA), Athens, Greece,
[2] Institute of Computer and Communications Systems (ICCS), Greece

Abstract. Recommendation engines are widely used in order to predict the rating that a user would give to an item based on the user's past behavior. Modern recommendation engines are based on computational intensive algorithms like collaborative filtering that needs to process huge sparse matrices in order to provide efficient results. This paper presents a novel scheme for the acceleration of Alternating Least Squares-based (ALS) collaborative filtering for recommendation engines that can be used to speedup significantly the processing time and also reduce the energy consumption of computing platforms. The proposed scheme is implemented in reconfigurable logic and is mapped to the Pynq platform that is based on an all-programmable MPSoC Zynq system. The hardware acceleration is integrated with the Spark framework and evaluated on real benchmarks from movielens. The performance evaluation shows that the proposed scheme can achieve up to $120x$ kernel speedup and up to $12x$ energy-efficiency compared to the embedded ARM processor of Zynq.

Keywords: reconfigurable logic, recommendation systems, cloud computing

1 Introduction

The need for large scale and energy efficient computation has evolved to unprecedented levels. Huge amounts of data are being gathered from multiple sources [1] such as social networks, IoT devices and web pages in general, while simultaneously we are deploying more sophisticated algorithms to process this data in order to perform various AI and machine-learning tasks. Big organizations offering such services, like Google, Amazon and Microsoft are constantly expanding their data-center infrastructures to meet the processing demands. However traditional semiconductor technologies are reaching their physical limits [2] and the Moore's Law seems unable to back up this challenge. Moreover energy consumption is becoming the dominant limiting factor in datacenters. In order to meet the needs for huge processing power and energy efficiency, novel architectures based on reconfigurable logic are adopted by data center operators.

This paper presents a novel scheme for the acceleration on recommendation engines that are based on collaborative filtering. The proposed system is evaluated using the Pynq boards built around the Zynq-7000 platforms. Zynq is an Heterogeneous Soc that incorporates both an ARM CPU and a FPGA [3].

The main contributions of this paper are the following:

- Design Space Exploration of a Recommendation System using Matrix Factorization trained by Alternating Least Squares.
- Efficient mapping in reconfigurable computing using High-Level Synthesis (HLS).
- Performance and power evaluation.
- Creation of a python interface for the accelerator.
- integration with the Spark framework through python.

2 Related Work

On 2009 D. Yang et al. presented an FPGA Implementation for Solving Least Square Problem [11]. However, at the time the urge for low energy consumption was not as intense as today and as a result there was no reference to power metrics. Moreover the design took place on a an FPGA and not on SoC embedded heterogeneous platform.

In this paper we present a novel approach focusing both on performance and power consumption of the recently released Zynq device, for alternating least squares learning algorithm which is an extension of least squares algorithm, with the second one having somewhat more applications in modern computing. Our prototype cluster is almost identical with the one presented here [14] and here [15], except from the fact that we used the Pynq Boards instead of ZedBoards and Apache Spark, instead of Hadoop. Furthermore in our case the bitstream can be downloaded at runtime, as long as it is stored in the boards SD card, using a module that comes with Pynq's image. A very nice proposal concerning the integration of FPGAs in data centers is presented here [16], but emphasis is given in the aspect of partial reconfiguration which is not considered at all in this paper. A more related work to this paper is [12] on Neighborhood based Collaborative Filtering on Zynq (2015), however in this paper we also present an attempt to integrate the kernel with a high level language on cluster running Apache Spark. Work related to accelerators running in parallel on a cluster has taken place in from Muhuan Huang at UCLA [13].

3 Algorithm Overview

For this study we developed a recommendation system in software, which uses collaborative filtering with matrix factorization and is trained with the Alternating Least Squares (ALS) learning algorithm. Without Loss of Generality we assume that the items to be recommended are movies.

3.1 Brief Algorithm Description

Let $R = [r_{ij}]_{n_u \times n_m}$ denote the input user-movie matrix, where each element r_{ij} represents the rating score of user i to movie j with each value being either a real number or missing, n_m and n_u denotes the number of movies and number of users respectively. Our task is to fill the missing values of R with values as close to reality as possible, based on the known values.

Both movies and users are modeled with a feature vector and each rating (either known or unknown) as the inner product of the corresponding movie and user Vector. Let $U = [u_i]$ be the user feature matrix where $u_i \in \mathbb{R}^{n_f}$ for $i = 1...n_u$, and let $M = [m_j]$ be the movie feature matrix , where $m_j \in \mathbb{R}^{n_f}$ for all $j = 1...n_m$. The dimension of the feature space in n_f, it is the number of features/latent-factors the algorithm will have to learn for each user and each movie. Determining the best possible n_f as well as some other model regularization parameters ,which will be presented later on, can be achieved by cross-validation or other popular techniques used in machine learning, but at this study we won't focus at all on the methods used to find the optimal values for the mentioned variables.

Ideally we would like to achieve $r_{ij} =< u_i, m_j > \forall i, j$. In practice however we try to minimize a loss function of U and M to obtain them. In this algorithm we examined the Mean-Squared-Error however our purpose is not to tune the model-parameters as best as possible to minimize RMSE but to accelerate the algorithm. The loss function due to a single rating is as follows.

$$L^2(r, u, m) = (r - < u, m >)^2 \tag{1}$$

Then the total loss function,given the whole matrices U, M can be defined as the average loss on all known ratings

$$L^{total}(R, U, M) = \frac{1}{n} \sum_{(i,j) \in I} L^2(r_{ij}, u_i, m_j) \tag{2}$$

where I is the index set of all known ratings and n is the number of elements inside I.

Our algorithm's task is as follows

$$(U, M) = min_{(U,M)} L^{total}(R, U, M)$$

where $U \in \mathbb{R}^{n_u \times n_f}$ and $M \in \mathbb{R}^{n_m \times n_f}$.
Hence, we have totally $(n_u + n_m) \times n_f$ free parameters that are used for the learning process. We avoid over-fitting by using Ttkhonov regularization [5] term to the total cost function.

The algorithm we used for acceleration [5] can be summarized in the following steps

1. Initialize matrix M by assigning small random numbers to the movie vector elements.

Table 1: Execution time as deducted by 10 iterations on the movielens-1m dataset.

Operation	Execution Time(%)
Matrix_Op	92.35
Cholesky	7.02
Rest	1.63

2. Fix M, Solve U by minimizing the objective function (the sum of squared errors);
3. Fix U, solve M by minimizing the objective function similarly;
4. Repeat Steps 2 and 3 until a stopping criterion is satisfied.

Let I_i denote the set of movies j user i has rated and I_j denote the users that have rated movie j.($\mathbf{card}(I_i) = n_{u_i} \quad and \quad \mathbf{card}(I_j) = n_{u_j}$)

$$u_i = A_i^{-1} V_i \forall i \tag{3}$$

where $A_i = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E, V_i = M_{I_i} R(i, I_i)$ and E is the $n_f \times n_f$ identity matrix. Once again M_{I_i} is the sub-matrix of M where the columns $j \in I_i$ are chosen, and $R(i, I_i)$ is the i'th row of R from which only the columns $j \in I_i$ are chosen.
Similarly, in case we want to update the elements of M each m_j is calculated by using the feature vectors of the users who have rated the corresponding movie j and of course the ratings themselves:

$$m_j = A_j^{-1} V_j, \forall j \tag{4}$$

in this case, $A_j = U_{I_j} U_{I_j}^T + \lambda n_{m_j} E$ and $V_j = U_{I_j} R(I_j, j)$. U_{I_j} is the sub-matrix U where only rows $i \in I_j$ are chosen, and $R(I_j, j)$ is the sub-vector of R where only rows $i \in I_j$ of the j'th column are chosen.

4 Profiling Execution Time

The first step of our design methodology was to profile the algorithm in order to indicate the most computational intensive part. We executed the algorithm with input movielens_1m [9] and number of features ranging form 10 to 100 with a step of 10, with ten iterations executed for each latent factor we found out that the execution profile converges approximately to that presented in Table 1. With the matrix operations occupying approximately 92% of the execution time we proceeded with the design of such a kernel.

5 Prototyping on Zedboard using SDSoC

5.1 Data Mapping

To achieve an efficient hardware implementation, the memory access pattern was studied, in order to map input and output data in a way that prevents

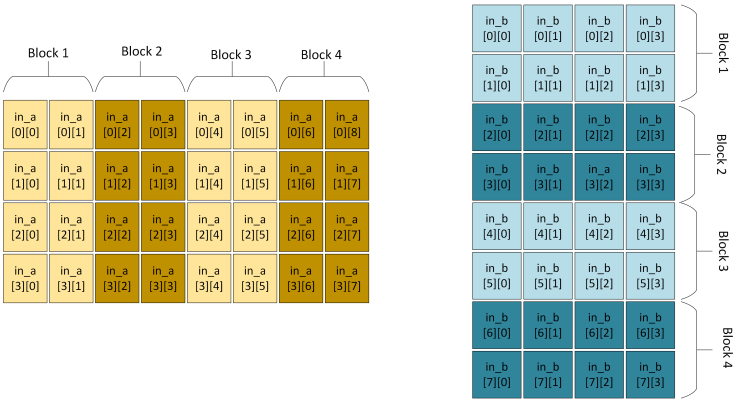

Fig. 1: Input data mapping on Brams. With this type of mapping we can fetch whole columns or whole rows in a clock cycle.

bottlenecks. Zynq's programming logic features dual port BRAMs meaning that we can fetch two elements per cycle from each. Hence, we partitioned the data in pairs of rows or columns, depending on the memory access pattern of each calculation type.

5.2 Computational Part of the Kernel

For the implementation of the kernel, we developed two similar computational units that work in parallel in order to simultaneously perform operations on four matrices. Each computational unit consists of a DSP48 row that is fed with raw input data. The DSP48 row performs multiplications in parallel and feeds the result in a tree adder. The whole unit is pipelined in order to maximize the throughput.

5.3 Software - Kernel interface version 1

In the first version of the kernel we used AXI4-Stream to transfer the data between the DMAs and the hardware function. For this version we designed the protocol's interface directly on the accelerator. In this version a software driver was used for sending data windows of size 20×80 to the kernel. Although this implementation achieved great speedup against the ARM-only execution, it had many drawbacks like the need for two-dimensional zero-padding of the input data, and the need to transfer multiple arrays used as intermediate accumulators for the storage of the partial results.

5.4 Software - Kernel interface version 2

In an attempt to increase the performance we used Xilinx's IP FIFO Accelerator Adapter. This IP is responsible for managing efficiently the AXI4_STREAM

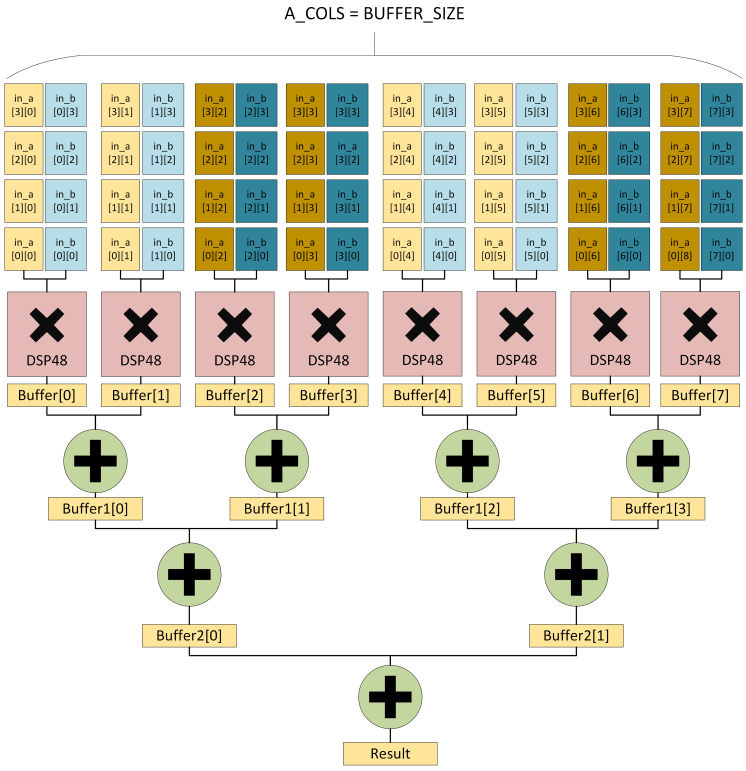

Fig. 2: Abstract representation of the main computational unit. The DSP48 row is longer in the actual implementation.

protocol exposing to the hardware kernel a simple FIFO interface. This IP relieved us from the need to design our own AXI4-stream interface and increased even more the performance, but didn't solve the problems emerged from the first version of the interface.

5.5 Software - Kernel interface version 3

In the final version we took advantage of the Accelerator Adapter's programming capabilities to boost up the performance. More specifically we configured the adapter in a way that allows passing arrays, from software to hardware, whose one dimension is determined at runtime from the processing system, and as a result we reduced the amount of padding needed to the data transferred, leading to less unnecessary operations which in turn leads to greater performance and energy efficiency.

6 Python Integration on Pynq

To leverage the use of the hardware acceleration unit and make easy the utilization from high-level programming framework, we developed the required APIs that allow the transparent deployment of the accelerators. Specifically, we developed the required libraries that allow the instantiation of the kernel from a high-level language like python, which is widely used in Machine Learning Tasks. The whole process took place by using the Pynq Board, which is a prototype board from Digilent that comes with a Linux image containing python libraries that help designers use kernel's from python scripts. The whole process is described below:

1. We created a bitstream for our IP matching the new Device(PYNQ), using Vivado.
2. Then we wrote the software Part of the algorithm in Python, using efficient libraries (numpy, scipy).
3. Finally by using the libraries coming with the Linux image of PYNQ, we created the appropriate software driver responsible for the software-hardware communication.

At the final step of the mentioned process we had to perform manually the operations that are performed by SDSOC framework automatically. The Python Libraries are wrappers of C language that are used for the interprocess communication. This wrap is accomplished with the use of a library called *cffi*, which allows python scripts to execute C code coming either precompiled either in source-code form. This means that this integration can happen in any platform rather than Pynq. Moreover with the use of *cffi* we can hide low level implementation details from the developer under python hood.

7 Apache Spark Integration

Apache Spark [6] is a framework designed for fast large-scale data processing. Spark stores data in a structure called resilient distributed datasets (RDD) [7], that is a read only (for ease of coherency purposes) collection of the data. Spark data operations are scheduled in a DAG scheme. Each task consists of a series of **transformations** that generate new RDDs and an **action** which corresponds to the reduce step of the map-reduce programming model. Spark performs lazy evaluation, in order to perform as much tranformations as possible in one step so that it can achieve more efficient task scheduling. In a glance it is an improved version of Hadoop MapReduce [8]. At the moment, Spark is one of the most popular big-data frameworks.

On this step we made a prototype Cluster consisting of four Pynq boards in order to run the algorithm both in parallel using Apache Spark and accelerated using the programming logic of each PYNQ [10]. The idea is that every worker of the cluster, each PYNQ board in our case, contains the bitstreams of the accelerator and the Apache driver program commands the workers to configure

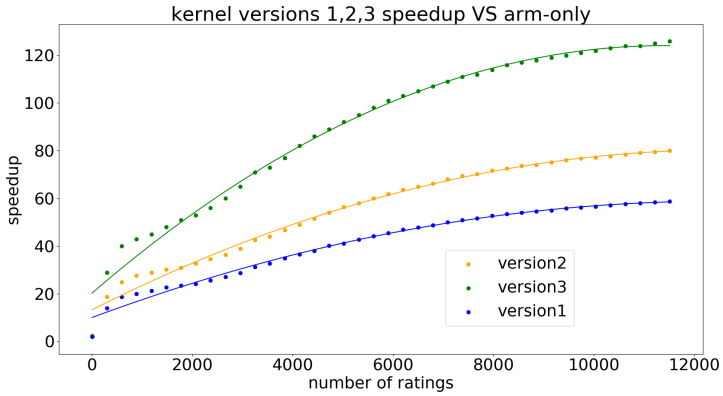

Fig. 3: Accelerator speedup against arm only execution. Points represent our measurements for different input sizes. $n_f = 80$

their FPGAs appropriate for the computation that is about to happen. This happens by calling a dummy *map*() function,before the actual computational *map*() operation, whose purpose is to instruct the workers to overlay the appropriate bitstream.

8 Performance evaluation

8.1 Kernel-only Performance Evaluation on Zedboard

The first implementation created on SDSoC framework, achieved speedup of up to 120× for input matrices of size 12000×80, against the arm-only execution. As the input matrix size increased, the speedup was also increased. It is important to notice that Figure 3 refers only to the speedup of the accelerated part (kernel) and not the speedup of the whole ALS algorithm.

8.2 ALS performance Evaluation Zedboard

Embedding the Version 3 kernel in ALS algorithm and running iterations using the datasets movielens_1m and movielens_100k we get the speed_up shown in Table 2. Notice that the column presenting the average number of ratings per movie/user is present, because this is a good metric indicating the average size of the matrices that will be produced at runtime. As a result, from this metric combined with Figure 3 and Amdahl's law, we can estimate the anticipated speedup for the specific dataset, this observation is actually verified by the actual speed-up measurements which happen to be very similar to the ones anticipated.

We also compared this implementation against a software only implementation on two other platforms, an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz,

and Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz. For this comparison we used as input the movielens_1m dataset and the result was that the implementation on the accelerated embedded system outperformed the i7 processor by a factor of 1.7× and the Xeon processor by a factor of 2.7×.It is important to notice that such small datasets like movielens_1m are unable to demonstrate the kernel's full potential which is presented in Figure 3, because the small number of ratings per user/movie leads to matrix operations of very limited size.

Table 2: Execution time speedup as deducted of the ALS algorithm on datasets movielens-1m and movielens_100k with $n_f = 80$.

Dataset	Speed-up vs Arm-only	average ratings per movie/user
movielens_100k	18.8×	76.2
movielens_1m	36×	205.2

8.3 Power Consumption

In Figures 4,5 we show the power consumption of the algorithm for one iteration on two different datasets. Although the accelerated version consumes more power momentarily in the beginning of the execution the fact that it runs for much less duration leads to a great improvement to the Performance per Watt metric versus the arm only execution. Specifically one iteration on movielens_100k dataset consumed 12× less energy while an iteration on movielens_1m consumed 27× less energy. We can notice both in performance and energy consumption evaluation that the kernel scales very well, meaning that as the input size increases the performance speedup and the energy savings increase too.

Table 3: Energy savings as deducted of the ALS algorithm on datasets movielens-1m and movielens_100k.

Dataset	Energy savings	average ratings per movie/user
movielens_100k	12×	76.2
movielens_1m	27×	205.2

8.4 Python on Pynq

This implementation showed-up great results in performance but the speedup achieved compared to an arm only execution was quite reduced compared to the one achieved on the previous implementations. The reason is that a high

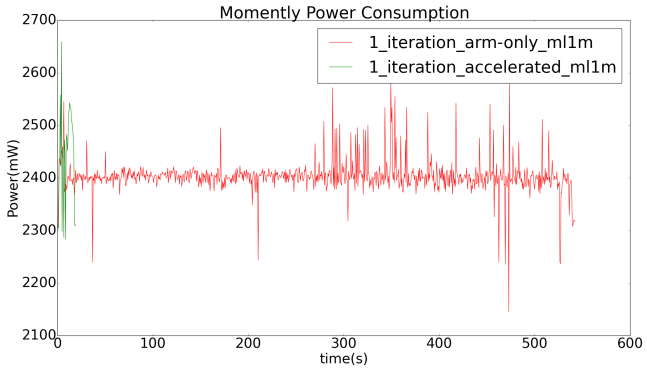

Fig. 4: Power consumption profiling of the system for 1m dataset (from one iteration)

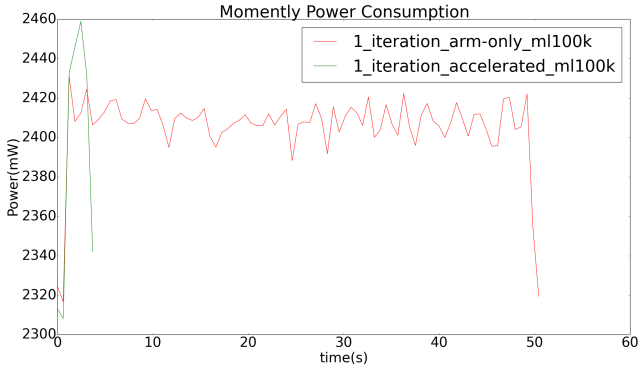

Fig. 5: Power consumption profiling of the system for 100k dataset (from one iteration)

level language like python and and its corresponding libraries were not created having in mind integration with hardware accelerators and as a result specific data conversions are needed that consume great percent of the execution time.

8.5 Apache Spark Integration

Four Pynqs accelerated and coordinated by spark managed to run 4-5× faster than an arm only execution. On this case there are many software parts added to the algorithm by spark. Spark adds serialization and deserialization tasks data broadcasts over Ethernet and more. As a result the part which is accelerated is smaller compared to the total execution time and as a direct impact of Amdahl's law we expected a smaller speedup.

Table 4: Execution time speedup as deducted of the ALS algorithm with $n_f = 80$ on datasets movielens-1m and movielens_100k.

Dataset	Speed-up of Python implementation
movielens_100k	5.8×
movielens_1m	13.8×

9 Conclusion and Future Work

In this paper we discussed the path of reconfigurable architectures as an computational alternative path, and we attempted to sum a performance and energy evaluation of that path on embedded boards. Moreover we attempted to test this technology with a popular scripting language like python in order to make it more accessible and easy to use by software developers. The results are definitely promising from both power consumption and performance perspectives. However in order to make these architectures a common case, we must expand the library so that it contains multiple accelerators, for many computational intensive tasks. Moreover it is great need to make these accelerators easy to use by constructing a fine tuned and efficient python library that allowed smooth transitions from software execution to hardware and vice versa. Except from python, apache spark could be extended in order to natively support accelerated execution more efficiently, by integrating specific instructions for configuring the slave's programming logic instead of forcing us to use "map()" calls that don't have computational intentions but were just written for FPGA configuration purposes.

Acknowledgment

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 687628 - VINEYARD: Versatile Integrated Heterogeneous Accelerator-based Data Centers.

References

1. PK Gupta, Director of Intel Cloud Platform Technology,Xeon+FPGA Platform for the Data Center (2015)
2. Hadi Esmaeilzadeh,Dark Silicon and the End of Multicore Scaling, ISCA (2011)
3. Vidya Rajagopalan,Xilinx Zynq-7000 EPP An Extensible Processing Platform Family (2011)
4. Yahuda Coren,Matrix Factorization Techniques For Recommender Systems, publisher IEEE Computer Society (2009)
5. Yunhong Zhou,Large-Scale Parallel Collaborative Filtering for the Netflix Prize (2008)
6. Matei Zaharia,Spark: Cluster Computing with Working Sets,Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010 (2012)

7. Matei Zaharia,Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (2012)
8. Juwei Shi , Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics , Proceedings of the 41st International Conference on Very Large Data Bases, Kohala Coast, Hawaii (2015)
9. Xiang Ma, Chao Wang, Qi Yu, Xi Li, Xuehai Zhou, An FPGA-Based Accelerator for Neighborhood-Based Collaborative Filtering Recommendation Algorithms, Cluster Computing (CLUSTER), 2015 IEEE International Conference on, September, 2015
10. Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris, Spynq: FPGA acceleration of Spark applications in a Pynq cluster, FPGA acceleration of Spark applications in a Pynq cluster, IEEE nternational Conference on Field-Programmable Logic and Applications, September, 2017. Ghent Belgium
11. Depeng Yang, An FPGA Implementation for Solving Least Square Problem, IEEE (2009)
12. Xiang Ma, An FPGA-based Accelerator for Neighborhood-based Collaborative Filtering Recommendation Algorithms, Cluster Computing (CLUSTER), IEEE International Conference (2015)
13. Muhuan Huang,Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale ,SoCC Proceedings of the Seventh ACM Symposium on Cloud Computing (2016)
14. Zhongduo Lin, Paul Chow, ZCluster: A Zynq-based Hadoop Cluster, IEEE, pp. 450-453 (2014)
15. Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, Avesta Sasan, and Houman Homayoun, Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGAs, IEEE International Conference on Big Data, pp. 115-123(2015)
16. Fahmy, Suhaib A., Vipin, Kizheppatt and Shreejith, Shanker, Virtualized FPGA accelerators for efficient cloud computing. IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Vancouver, Canada, 30 Nov - 3 Dec pp. 430-435 (2015).