Towards a Low-Cost FPGA Micro-Server for Big Data Processing

Mohamed Abouzahir[1]
Ecole Supérieure de Technologie de Salé
Laboratoire LASTIMI
Université Mohammed V de Rabat

Khalifa Elmansouri[2]
Institut Supérieur des Sciences de la Santé (ISSS)
Laboratoire des Sciences et Techniques de la Santé
Université Hassan 1er, Settat

Rachid Latif[3]
Ecole Nationale des Sciences Appliquées d'Agadir
Laboratoire d'Ingénierie des Systèmes et Technologies de l'Information (LISTI)
Université Ibn Zohr, Agadir

Mustapha Ramzi[4]
Ecole Supérieure de Technologie de Salé
Laboratoire LASTIMI
Université Mohammed V de Rabat

***Abstract*—The development of big data in the era of data explosion and the growing demand for micro-servers in place of traditional servers to adapt to lightweight tasks in recent years has put into question how to integrate and make use of these two important domains. During the same era, CPU performance growth has reached a certain maturity. In order to surpass these issues and to reach high performances computing, a new trend now is to use multiple processing units or heterogeneous components in micro-servers to reduce computational complexity. The implementation of Big Data processing algorithms using embedded heterogeneous architectures rises a new challenges due to constraints of the used architecture-based system on chip which require a special attention and imposed new demands to our works. In this article, we focus on using embedded FPGA accelerator to give a solution to this problem. Precisely, we will attempt to prototype a micro-server for the processing of big data on FPGA and compare its performances with a high-end GPGPU using existing benchmarks. The implementation on the FPGA is done using a High-Level Synthesis based-OpenCL (HLS) instead of the traditional description language. The obtained results shows that FPGA is an interesting alternative and can be a promising platform to design a micro-server when it comes to process a hug amount of data, in particular with the emerging technologies for FPGA programming using HLS approach and by adopting the OpenCL optimization strategies.**

Keywords*—Arria 10 FPGA (Field Programmable Gate Arrays); GPGPU (General Purpose Graphics Processing Unit) ; big data; parallel computing; (HLS) High-Level Synthesis

I. Introduction

BIG data, produced from online transactions, emails, videos, audios, picture, posts, search interrogation, medical records, social networking interface and science applications, has become one of the most important domain in the information technology industry [1]. According to IBM (2015), around 2.5 nonillion (10 to the power of 30) bytes of data is created every day. The overwhelmingly large amount of data leads to challenges including storage, analysis and fetch. To define the properties of big data, Doug Laney (2001) has proposed the 3Vs (Volume, Variety, Velocity) [2]. In current hypercompetitive industry environment, several challenges related to big data processing are imposed on the companies, which include to meet with the need for speed, to understand the data, to address the data quality, to display significant results and to distinguish the outliers. There is growing interest in FPGA (Field Programmable Gate Array) as a solution. FPGA has always been considered as the generation of integrated circuit that could replace ASIC (application specific integrated circuit). Able to be fully reconfigured by user, an FPGA is commonly claimed higher performance, shorter time to market, lower cost, high reliability, less needs for long-term maintenance. Being reprogrammable results is the main difference between an FPGA and an x86 processor: the FPGA does not waste compute cycles doing unnecessary processing. In other words, FPGA excels for doing one simple and repetitive task like pattern matching. This great advantage of FPGA has catalyzed investigations on its potential usage in big data processing. One of the most important assets of FPGA is its exceptional ability in the computation of finegrained tasks in clock-cycle basis. This ability is extremely interesting especially regarding Big Data management. However, this high-potential acceleration constrains drastically the possibilities of implementation. After having chosen the FPGA design, the aim is to reproduce this precise organization of the blocks matrix using OpenCL code and the IntelFPGA OpenCL SDK, which generates a board organization and the full setups given an OpenCL code. In this article, we investigate the challenges for speeding up Big Data algorithms and provide a roadmap for further improvement. Ultimately, our aim is seamless integration of FPGAs to build a micro-server for Big Data processing. Many recent research has been conducted to strengthen the niche of FPGA in big data business [3], [4], [5]. However, the current available tool sets to build FPGAs are still complex. On the other hand, GPUs are leveraged well with the implementation of CUDA language. Besides, GPUs where instructions are executed in fixed instruction set are more flexible; they are also more adapted for floating point arithmetic.

The paper is organised as follow: Section II present the related work and our contribution, Section III presents our methodology applied for performance evaluation. In Section IV, we will give a description of the test benchmarks implemented in our work. Section V presents the hardware specification as well as the software tools and the adopted parallel programming technology. In Section VI we will present the algorithms implementation as well as the performance evaluation. Section VII gives a holistic overview and conclude the work

II. Related Work and Contribution

Authors in [6] presented an FPGA-Accelerated Big Data implementation. The proposed system is based on the popular Apache Spark framework on the software side, and on an OpenCAPI-based POWER9 platform with Xilinx VU37P FPGA on the hardware side. Their system is able to generate a high-performing FPGA circuit from very high-level code descriptions in Spark.

The work in [7] present a case study of accelerating Apache Spark using re-configurable architecture. The authors proposed a framework to integrate FPGA accelerator into a Spark cluster. The Spark tasks are accelerated on the FPGA using Python. The performance results are evaluated with a case study of 2D FFT algorithm acceleration. The obtained results showed that FPGA based Spark implementation acquires 1.79x speedup than a conventional CPU implementation.

The author in [8] proposed the use of distributed databases and high-performance computing architecture in order to exploit multiple re-configurable computing and application specific processing. The proposed a 4-layer general architecture for smart agriculture, which is able to collect, store and process data from IoT nodes, integrate external data from other sources and allows efficient treatments of data coming from several sources with a cloud high-performance heterogeneous architecture.

A collaborated research team from George Mason University and University of California, guided by Netshatpour [3] has discovered a significant speedup with respect to K-means, KNN, SVM and Naive Bayes while implementing the mentioned machine learning algorithms in a Hadoop Platform with Intel Atom C2758 and Xeon E5 as master nodes and several Xilinc Zynq devices as slave nodes.

Besides, [4] have presented an FPGA-based hardware accelerator platform for big data matrix processing. A comparison of performance has been conducted between an Intel I7-4770 CPU (3.4GHz) and an FPGA of the model VC707 (125 MHz).
Furthermore, a recent research published by the University of Science and Technology of China [5] has presented a software-defined operating system framework for FPGA based accelerator with the implementation on Xilinc Zynq FPGA.

On the other hand, ITRS Semiconductor roadmap foresees that hundreds of processors would be the base for the next generation embedded multicore designs. Recently, Microsoft in 2015 collaborated with Bing to investigate the use of FPGA. The project, also known as Project Catapult, has showed an improvement of nearly a factor of two of the operations per second in a critical component of Bing search engine [9].

With the demand for high speed network and computing, speed and parallel algorithms have become essential tools for development. Many of these operations were performed by a general purpose processor. But now days due to the availability of FPGAs, many researchers try to implement various algorithms on FPGAs more efficiently. FPGAs are often used as hardware accelerators. Our work aimes to implement big data benchmarks and to compare the performance of GPUs and FPGAs when it comes to big data processing. The main idea of our targets is to construct a scaled up platform of FPGA-based micro-server for big data processing. To date, among all the current research and works in the state of the art, none of them targets the application of FPGA in big data processing. To our knowledge, this is the first work to evaluate and optimize big data algorithms on a dedicated architecture by adopting the high level synthesis approach.

TABLE I. Performance Metrics

Metrics used for FPGA and GPU	Additional metrics for FPGA
Execution time	% of DSP blocks
Memory bandwidth	% Logic Elements
-	% of Memory Block

III. Performance Evaluation Methodology

A. Performance Metrics

The FPGA and the GPU we plan to use does not have the same I/O maximum speed, which will result in a comparison error if the slowest computing unit is limited by its I/O maximum bandwidth. Therefore, for each algorithm and each implementation, we plan to save the different metrics during the computing of the same series of test files on the two components. These metrics will then be compared by being put into charts and analysed.

For the GPU: We use Nvvp which is a built-in tool of Nsight to evaluate the performance of GPU. This tool allows us to measure the running time and the throughput of each function implemented, hence, allows us to know which parts of the code can be improved. For FPGA: Assuming that there is still no benchmark available regarding FPGA for the algorithms we decided to propose our own implementation. There is no implicit way of evaluating their performances precisely. Therefore, our study is based on a data-processing speed comparison between an FPGA and a GPU, the last being already used for massive parallel computing. The performances of our algorithms will also be verified by the IntelFPGA SDK for OpenCL, which includes an optimization report. The Table I gives some important metrics evaluated in our results.

B. Multiobjective Optimization and Pareto Optimality

FPGA programming is about finding algorithms that optimize some aspects of the performance regarding different limited resources. Therefore, there is no unique solution to those problems, and the optimal computation depends on the factors that we want to optimize. No single FPGA implementation for all benchmarks can be an optimal <P, E, A> with P for Performance, E for Energy and A for Chip Area [10]. Therefore, with N benchmarks each being potentially individually implemented Pareto like with numerous configurations the automatic optimization variations on concurrency E against A is needed.

TABLE II. EXECUTION OUTPUT AND DATASET STATISTICS

Average path distance	3.692507
Network diameter	8
Global efficiency	0.306578
Clustering coefficient	0.632353
Transitivity	% 0.000073
Betweenness centrality	4051.734470
Closeness centrality	0.261441
Degree Distribution	0.023026
Pearson correlation coefficient	-1.536665

C. Graph Measures Results

The implemented big data algorithms were tested on the Stanford Large Network Dataset Collection provided by SNAP (Stanford Network Analysis Project) [11]. We have executed our algorithms on the dataset consisting of <circles> (and <friends lists>) from Facebook, which can be represented by an undirected graph of 4039 vertices and 88234 edges. Table II represent the execution outputs of the created graph. We have implemented the graph using C language and produce the algorithms to measure the dataset statistics: the Average path distance, Network diameter, Global efficiency, Clustering coefficient, Transitivity, Betweenness centrality, Closeness centrality, Degree distribution and Pearson correlation coefficient.

D. Test-bed Setup

To evaluate the OpenCL FPGA implementation, we used a host computer, operating at 2.5 GHz under CentOS Linux 7.0 with a 32 GB RAM, with FPGA board mounted on the PCIe slot. We used the De5a-Net board embedding the IntelFPGA Arria 10, Fig. 1 shows the test-bed setup.

IV. ALGORITHM DESCRIPTION

A. K-means

K-means is a popularly-used algorithm for clustering. The aim of clustering is to divide the given set of data X composed of n points into partition $\{C_i\}_{1<i<k}$ such that points in each subset are similar to each other; otherwise, points from different groups are dissimilar [12]. The similarity is defined by a distance function; therefore, clustering task is able to be interpreted quantitatively as minimizing the cost function desired by user, which is normally formulated as below:

$$e_k(X, C) = \sum_{i=1}^{n} \min(D(x_i, c_j)) \tag{1}$$

where c_j is the center of subset C_j

For K-means, we set the distance $D(x, y)$ by the square of Euclidean distance $\|x-y\|^2$ (this is not a metric, because it does not have triangular inequality property). Hence, K-means cost function is defined by:

$$e_k(X, C) = \sum_{i=1}^{n} \min_{1<k<j} \|x_i - c_j\|^2 \tag{2}$$

Given a set of data points, a clustering algorithm aims to the similarity which is defined using distance measure. In our work, the Euclidean distance is utilized. Basically, we start by choosing K points called centroids randomly among the given points then form K clusters, each of which contains a centroid and the points that accept this centroid as the nearest one. We gradually update these centroids by calculating the center of mass in each group. This algorithm terminates when the number of iterations exceeds a chosen number or the change after each iteration is less than a chosen threshold

1) Lloyds heuristic algorithm for Kmeans: The Lloyds heuristic algorithm [13] for clustering high-dimensional data is usually described by 4 steps. Firstly, stop condition is defined as following: algorithm terminates after exceeding a number of loops or whenever the difference of $e_k(X; C)$ between two consecutive loops is less than a real positive threshold r given.

- Step 1: Initialize k temporary centroids. Start loop of N iterations:
- Step 2: For each x in given data set, search for the nearest centroid c from x and assign x to this cluster.
- Step 3: For each cluster C_i, calculate the new centroid of C_i by following formula :
$$c_i = \frac{1}{C_i} \sum_{x \in C_i} x \tag{3}$$
- Step 4: Calculate the new value of $e_k(X; C)$, then the difference $\Delta = e_k(X; C)^{new} - e_k(X; C)^{old}$. If Δ is smaller than r, return the contemporary assignment and end the loop.

2) Initialization Method:

a) Method 1 (Forgy [14]): : Choose arbitrarily k points from data set, this method gives us no guarantee about how close the cost function will be to the global minimum. Therefore, to increase the chance to get well-accepted result, we repeat this initialization l times and pick out which gives the best output.

b) Method 2 (K-means++): : This method guarantees that:

$$E[e_k] \in 8(2 + \ln k)e$$

where e is the global minimum, hence allows us to control the performance of heuristic algorithm.

Algorithm 1 K-means++

```
Choose $c_1$ uniformly from data set: $C++ = \{c_1\}$
for $i = 2$ to $k$ do
    Choose $c_i = x \in X$ with the probability
    $p(x) = \frac{D^2(x, C++)}{\sum D^2(y, C++)}$
    $C++ \leftarrow C++ \cup c_i$
end
```

B. Sorting Network

One of the commonly used operations in high speed data processing is data sorting. A sorting network consists of two types of items: comparators and wires. The wires are

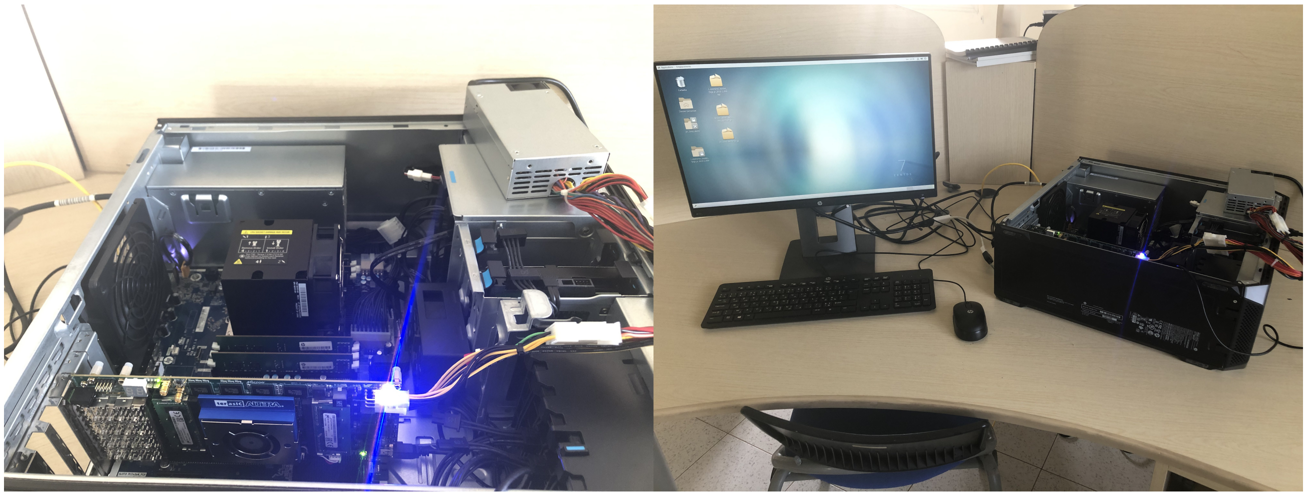

Fig. 1. Test-bed Architecture (DE5aNet board).

thought of as running from left to right, carrying values (one per wire) that traverse the network all at the same time. Each comparator connects two wires. When a pair of values, traveling through a pair of wires, encounter a comparator, the comparator swaps the values if and only if the top wire value is greater than the bottom wire value. Sorting networks differ from general comparison sorts in that they are not capable of handling arbitrarily large inputs, and in that their sequence of comparisons is set in advance, regardless of the outcome of previous comparisons. This independence of comparison sequences is useful for parallel execution and for implementation in hardware. Some well-known methods to construct a sorting network can be listed such as Batcher odd-even merge sort [15], bitonic sort [16], Shell sort [17] and the Pairwise sorting network [18], whose depth efficiency is $O(log^2(n))$. Basically, the sequential sorting algorithm requires at least comparisons. To boost the performance on treating massive volume of data, some specified algorithms are chosen to be parallelized depending on their characteristics (data dependency, device architecture, methods of communication, network topology, etc.). The most commonly used sorting algorithm is Bubble sorting. For efficient and reduced operations implementation of sorting, [15] proposed a technique of sorting using sorting networks.

1) Bitonic Sort Algorithm: The bitonic sort [19] is a divide-and-conquer comparison sort usually implemented with recursion. Keys are first ordered into bitonic sequences and are then sorted using a bitonic merger. The number of comparators required can be reduced by a factor of $\log^2(N)$ by combining the perfect shuffle with bitonic sorting. This sort fits the SIMD (single instruction multiple data) model because it is readily implemented in hardware using a parallel sorting network. Given sufficient hardware, this sort is capable of achieving $O\left(\log^2(N)\right)$ performance. A sequential, recursive version of the bitonic sort with running time $O\left(N\log^2(N)\right)$ is used on the microprocessor. The FPGA implementation uses a visualized, parallel sorting network. Both implementations sort in-place and require the input key quantity to be a power of 2; however, on the FPGA, we were able to use the same SIMD controller to schedule keys for an eight input sorting network

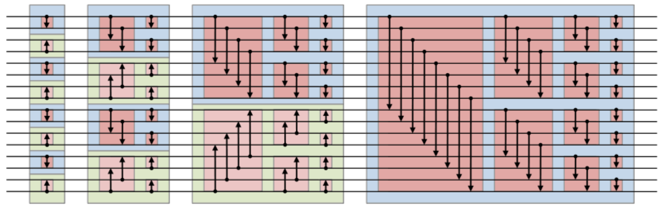

Fig. 2. Bitonic Sorting Network Illustration .

and a four input sorting network simultaneously. This allowed the use of all six memory banks. Bitonic sortings goal is to sort a bitonic sequence (a sequence (a_n) is called bitonic if and only if there exist an unique index i such that for all integer m, if $a_m; a_{m+1}; ...; a_{m+i}$ are monotonically increasing (or decreasing), then $a_{m+i+1}; ...; a_{m+2i}$ must be monotonically decreasing (or increasing)), which is easy to be parallelized and applied on hardware or software. For an arbitrary sequence, we can sort the first half of sequence in increasing order and the other in decreasing order, this transforms the sequence into a bitonic sequence in order to apply bitonic sort. For each i from 1 to $n = 2$, match two elements x_i and $x_{i+n/2}$ into a pair and swap them to form a pair in order (min; max). Consequently, we obtain two bitonic sequences whose lengths are a half of the given one (bitonic property is invariant by step 1) and also every element in the first half is smaller than every the second half. Apply recursively this procedure to each of two sub sequences until the length of sequence is less than 2.

C. Correlation Algorithms

Correlation algorithms such as Pearson [20] and Spearman [21] measure the dependence between variables. Popular applications may include calculating the relationship between age and number of hours spent watching TV, or the relationship between product sales and temperature. Assuming that comparisons and simple operations are both done in time $O(1)$, Pearson and Spearman correlations are done respectively in time $O(n)$ and $O(n\log(n))$. The Pearson correlation coefficient is given by the formula (4), assuming that a and b are two zero-mean real valuad random variables.

$$\rho(a,b) = \frac{E(a,b)}{\sigma_a, \sigma_b} \tag{4}$$

where $E(a,b)$ is the cross-correlation between a and b, and $\sigma_a^2 = E(a^2)$ and $\sigma_b^2 = E(b^2)$ are the variance of a and b respectively. It is more convenient to work with the squared pearson correlation coefficient given by (5)

$$\rho^2(a,b) = \frac{E^2(a,b)}{\sigma_a^2, \sigma_b^2} \tag{5}$$

The squared Pearson correlation coefficient give an insight about the strength of the linear relationship between two random variables. When $\rho^2(a,b) = 0$, then two random variables a and b are uncorrelated. When the value of $\rho^2(a,b)$ is near to 1, then a and b are said to be correlated. The squared Pearson correlation coefficient detects only linear dependencies between the two variables a and b. Indeed, If a and b are independent, then $\rho^2(a,b) = 0$, but the converse is not true. The Pearson correlation coefficient ρ_p is defined according to equation 6:

$$\rho_p = \frac{\sum_{i=1}^{N} a_i, b_i}{\sqrt{\sum_{i=1}^{N} a_i^2 \sum_{i=1}^{N} b_i^2}} \tag{6}$$

The Spearman correlation coefficient ρ_s is calculated in the same manner as ρ_p, except that ρ_s is calculated after both a and b have been rank transformed to values between 1 and N (Equation 7). When calculating ρ_s, a fractional ranking is used, which means that the mean rank is assigned in case of ties. For example, suppose that the two smallest numbers of a are equal, then they will be both ranked as 1.5 ($frac[1+2]2$). A mean centering is first performed (by subtracting $N/2 + 1/2$ from each of the two ranked vectors).

$$\rho_s = \frac{\sum_{i=1}^{N} a_{i,r}, b_{i,r}}{\sqrt{\sum_{i=1}^{N} a_{i,r}^2 \sum_{i=1}^{N} b_{i,r}^2}} \tag{7}$$

V. Hardware Specification

A. Field Programmable Gate Arrays

1) Arria 10 Architecture: As a dedicated architecture, We used the Arria 10 FPGA as a target platform for our algorithm implementation Fig. 3. Arria 10 is one of the latest chip produced by IntelFPGA delivering the highest performance at 20 nm. Arria 10 FPGA is a low power embedded architecture up to 40% lower power than previous FPGAs generation. It allows up to 1500 GB/s floating-point operation with DSP blocks. The system clock is 100 MHz. The chip also includes a Dual-Core ARM operating at 1.5 GHz. Table III shows the available resources in terms of logic elements, DSP and memory blocks of the Arria 10 FPGA.

2) High Level Synthesis: The Arria 10 FPGA programming is done using OpenCL based High Level synthesis [22]. This is to achieve an efficient and fast parallel implementation of the algorithm on FPGA. OpenCL (Open Computing Language) is the first open, royalty-free, unified programming model for accelerating algorithms on heterogeneous systems. Based on C (C99), it supports four kinds of processing units: CPU, GPU, FPGA and DSP (digital signal processors). The real asset of this language is to use different units at the same time, processing them in parallel, and using each one of them for what it is the best. However, because of the total differences in processing algorithms between the different sorts of units, an OpenCL code has to be optimized for each device. The IntelFPGA SDK for OpenCL allows avoiding the traditional hardware FPGA development, which is too complicated for the use when it comes to high performance computing, in order to achieve a much faster and higher level software development flow. It includes multiple optimizations and can produce deep reports of the compilation and the code optimization. The IntelFPGA SDK for OpenCL requires the Quartus Prime (Pro version for Arria 10 board), also known as Quartus II, to function optimally. Fig. 4 shows the compilation process of OpenCL code for FPGA.

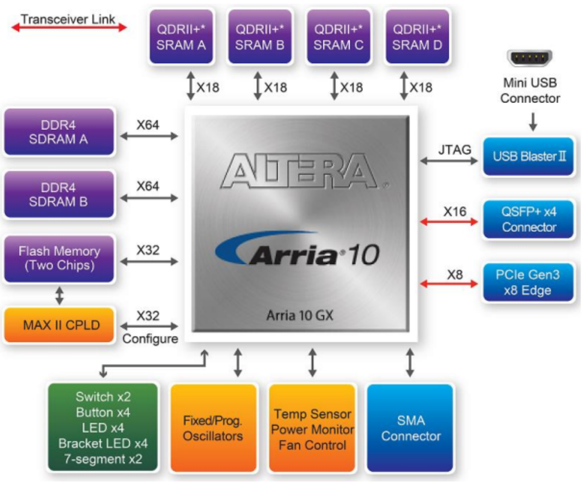

Fig. 3. Arria 10 Architecture .

TABLE III. Resources of Arria 10

Resource	Arria 10 device
	10AX115N2F45E1SG
Logic Elements (LE) (K)	1,150K
ALM	251,680
Register	1,006,720
Memory	32MB QDRII+ SRAM
	16GB DDR4 SO-DIMM SDRAM
DSP Blocks	1,518
18 x 19 Multiplier	3,036
17.4 Gbps Transceiver	48
PCIe Hard IP Block	4
Embedded memory	67-Mbits

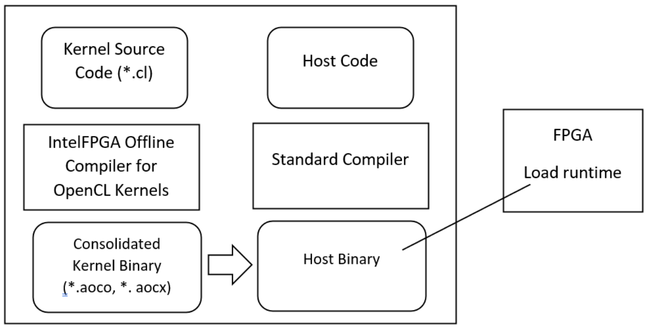
Kernel Source Code (*.cl)
Host Code
IntelFPGA Offline Compiler for OpenCL Kernels
Standard Compiler
FPGA
Load runtime
Consolidated Kernel Binary (*.aoco, *. aocx)
Host Binary

Fig. 4. The Flowchart of Compilation Process of the FPGA .

3) Quartus TCL Scripting: The Quartus II development software provides the scripting environment, particularly Tcl (tool command language) scripting. We use scripting support, to achieve custom analysis, automation and reproducibility. Custom analysis allows to build test procedures into the script and change design processing based on the test results. Scripts can automate design flows to perform on the computer and easily archive and restore projects. Reproducibility ensures that scripts use the same project setup and assignments for every compilation.

B. General Purpose Graphical Processing Unit GPGPU

1) GPU Architecture: In our work, we used the Nvidia GPU Quadro K2200 (Table IV). This GPU uses the first generation of Maxwell architecture released by Nvidia in February 2014 (the newest and second generation was released in September 2014). Maxwell introduces an all-new design for the Streaming Multiprocessor (SM) called SMM that dramatically improves energy efficiency compared to its predecessor Kepler. SMM uses a quadrant-based design with four 32-core processing blocks each with a dedicated warp scheduler capable of dispatching two instructions per clock. Each SMM provides eight texture units, one polymorph engine (geometry processing for graphics), and dedicated register file and shared memory. Maxwell improves on Kepler by separating shared memory from L1 cache, providing a dedicated 64KB shared memory in each SMM (for Quadro K2200). It provides native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap (CAS), which can be used to implement other atomic functions.

2) Programming the GPGPU: CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU virtual instruction set and parallel computational elements. The CUDA platform is designed to work with programming languages such as C, C++ and Fortran. Quadro K2200 uses the version CUDA 5.0. We are using Nsight Eclipse Edition (CUDA SDK 6.0) for algorithm implementation. This is a full-featured IDE powered by the Eclipse platform that provides an all-in-one integrated environment to edit, build, debug and profile CUDA-C applications.

VI. Algorithm Implementation and Performance Evaluation

A. FPGA Design and OpenCL Optimization

The conception of the design is a crucial step in the development process. We dispose of four types of blocks (Logic blocks, Memory blocks, Logic Register and Digital Signal Processing Blocks). Each one is able to perform a particular list of actions. Given these four sorts of blocks and their amount on the board, the aim is to associate to each one of them simple and statics instructions to do and to link them in a network. For instance, if we want to compute the sum of four 64-bytes integers i_1, i_2, i_3 and i_4, we chose to dedicate:

- One Logic Blocks programmed to do the summation of i_1 and i_2, then send the data directly to the third block.
- One Logic Blocks programmed to do the summation of i_3 and i_4, then send the data directly to the third block.
- One Logic block programmed to do the summation of the two previous results

On this example, we see an important characteristic of FPGA: the temporary variable of summation doesnt have to be stored, so there is no need for write/read operations, reducing the total amount of clock-cycles. If we want to be able to compute a new set of data every clock-cycle, the path taken by the data has to be the same regardless of the data themselves, even though we increase the number of comparison or assignment.

OpenCL is a language developed in order to support multi-platform computing. Considering the deep differences between computing units (CPU,GPU), the same OpenCL code is implemented highly differently on different platforms on a hardware level. Therefore, even though an OpenCL code for GPU will work on other platforms, it will only be optimized for GPU, and will certainly be highly inefficient if run on other computing units. It is the same for FPGAs: multiple valid implementations are inefficient, and there are multiple ways of coding that have to be promoted or avoided.

The first coding optimization specific to IntelFPGA OpenCL is the command #pragma unroll that has to be put in the OpenCL kernel file. Used before a loop, this command is read by the compiler, that process what is called the unrolling of the loop. On a hardware level, each iteration of a nonunrolled loop is by default done by the same area of the computing unit. The process of unrolling by a factor N consists in replicating the hardware $(N-1)$ time in order to be able to compute N iteration of the loop at the same time. If it seems efficient to unroll the loops, it is important to notice that the factor of the unroll has to be defined during the synthesis step, and cant be changed during the computation. Moreover, the hardware resources being limited, before unrolling it is important to be sure that the concerned loop results in a bottleneck of the overall computing process. It is useless to unroll a loop more than the number of iterations, and difficult to unroll it if the number of iterations is not easy to determine. It is important to underline the differences between the two kinds of parallelization that we have presented: the loop unrolling is a hardware-parallelization, whereas the ability

TABLE IV. Specifications for the Quadro K2200

	Processing Power (GFLOPS)	Memory Clock (MHz)	Memory			
			Size (MB)	Bandwidth GB/s	Bus type	Bus width (bits)
Quadro K2200	1280 (Single precision) 40 (double precision)	1250 (5000)	4096	80	GDDR5	128

of computing a new set of data every clock-cycle is a time-parallelization. Their combination can theoretically lead to drastic computing acceleration.
In order to simplify the unrolling process, it is important to avoid nested loops as much as possible. A nested loop is a loop called in another loop. The more loops inside a loop there are, the harder it is to unroll them. Therefore, an OpenCL code should promote one, maximum two loop levels with explicit amount of iterations. For these reasons, the OpenCL kernel for our algorithms are implemented with only two levels of loops having an explicit amount of iterations.

Another optimization possible to perform is the balance of reducing a set of data using an associative operation. For instance, if you want to add a set of N integers, the usual loop unrolled will use $N-1$ Logic Blocks. However, by doing a tree summation, the number of blocks can be reduced to $N/2$. This optimization can be managed by the compiler, even for more complex operations. Moreover, if the overall computation tree implies non-associative operations, the compiler can identify parts of the computation tree that can be balanced and balance them. Because of the difficulty in identifying such parts, the compiler proposes this optimization as an option. It is to the programmer to understand if this auto-balancing function is relevant, or to balance manually by modifying the code when the compiler cant extract the balancing. Many other constrains and way of coding are to consider when producing OpenCL code for FPGA, like the impossibility to use pointer to pointer parameter in the kernel functions, or the simplicity of indexes when arrays are called. In order to show the efficiency of OpenCL kernel optimization, we have implemented two version of Summary Statistics algorithm: Optimized (Algorithm 2) and unoptimized (Algorithm 3) kernels. This algorithm calculate the column-wise min, max, mean, variance, count, and number of non-zeros in a given dataset. (assuming each simple is done in O(1), all these statistics share the same complexity of O(n)).

Table V shows the estimated resource usage before kernel optimization. The problem reported by the optimization report is that too many kernels attempted to access the same variable at the same time (hereby is the variable sum), but there is only limited amount of access possible on the same variable each clockcycle. Therefore, the blocks that try to access have to wait, blocking the overall process and retarding it by N clock-cycle. An efficient way to avoid this problem is to use a shift-register with the size N, the maximum encountered late. The amount of clock-cycle is revealed by the optimizer.

Table VI shows the estimated resource usage after kernel optimization. The kernel optimization has improved the computation speed. The processing time is divide by a factor N, which is the number of clock-cycles that were lost because of blocking access. The IntelFPGA optimization report ensure to identify the most important bottlenecks and processes that slow down the computing and increase the number of clock-cycles taken by an overall computation. These optimizations strategy are adopted for the other algorithm in order to achieve an efficient parallel implementation with less resources usages.

TABLE V. Estimated Resource Usage Summary (Unoptimized Summary Statistics)

Resource	usage
Logic utilization	16%
Dedicated Logic registers	8%
Memory blocks	28%
DSP blocks	4%

Algorithm 3 Unoptimized OpenCL kernel

```
__kernel **void** summarystat (__global **float** *A, **unsigned int** size, __global **float***rep, __global **int***non_zero)

min = $A[0]$; max = $A[0]$; sum = 0.0f; sqsum = 0.0f; nonzero_count = 0;

#pragma unroll
for ($i = 1$; $i < size$; $i++$) do
    x = $A[i]$;
    sum $+= x$;
    sqsum $+= x * x$;
    if $x \neq 0$ then
        nonzero_count ++
    end
    if $x < min$ then
        min = x
    end
    if $x > max$ then
        max = min
    end
end
$rep[0]$ =max;
$rep[1]$ =min;
$rep[2]$ =sum/size;
$rep[3]$ =sqsum/size - (sum/size)$^2$;
$*$non_zero = nonzero_count;
```

B. OpenCL Implementations of Bitonic Sort Algorithm

If we consider a regular optimal sorting algorithms like the Quick sort, which has a $O(n \log(n))$ complexity in term of

Algorithm 2 Optimized OpenCL Kernel

```
__kernel void summarystat_optimized(__global float *restrict A, unsigned int size, __global float*restrict rep, __global int*restrict non_zero)
min_temp = $A[0]$; max_temp = $A[0]$; min_rep = min_temp; max_rep = max_temp; sum = $0.0f$; sqsum = $0.0f$; nonzero_count = 0;

float shift_reg_x_s[$N+1$]; float shift_reg_sqx[$N+1$];
float shift_reg_min[$N+1$]; float shift_reg_max[$N+1$];
int shift_reg_non_zero[$N+1$];
for $(i=0;\ i<N+1;\ i++)$ do
    shift_reg_x_s[$i$]=0;
    shift_reg_sqx[$i$]=0;
    shift_reg_min[$i$]=min_temp;
    shift_reg_max[$i$]=max_temp;
    shift_reg_non_zero[$i$]=0;
end
for $(i=0;\ i<size;\ i++)$ do
    x= $A[i]$; shift_reg_x_s[$N$]= shift_reg_x_s[0] + x; shift_reg_sqx[$N$] = shift_reg_sqx[0] + x * x; if $x <$ shift_reg_min[0] then
        shift_reg_min[$N$]=x
    end
    else
        shift_reg_min[$N$] = shift_reg_min[0]
    end
    if $x <$ shift_reg_max[0] then
        shift_reg_max[$N$]=shift_reg_max[0]
    end
    else
        shift_reg_max[$N$] = x
    end
    if $x \neq 0$ then
        shift_reg_non_zero[$N$] = shift_reg_non_zero[0] + 1
    end
    #pragma unroll for $(j=0;\ j<size;\ j++)$ do
        shift_reg_x_s[$j$]= shift_reg_x_s[$j+1$]
        shift_reg_sqx[$j$]= shift_reg_sqx[$j+1$]
        shift_reg_min[$j$]= shift_reg_min[$j+1$]
        shift_reg_max[$j$]= shift_reg_max[$j+1$]
        shift_reg_non_zero[$j$]= shift_reg_non_zero[$j+1$]
    end
end
#pragma unroll for $(j=0;\ j<size;\ j++)$ do
    sum +=shift_reg_x_s[$j$]; sqsum += shift_reg_sqx[$j$]; nonzero_count += shift_reg_non_zero[$j$]; if min_rep $>$ shift_reg_min[$j$] then
        min_rep = shift_reg_min[j]
    end
    if max_rep $>$ shift_reg_max[$j$] then
        max_rep = shift_reg_max[j]
    end
end
rep[0] = max_rep; rep[1] = min_rep; rep[2]= sum/size; rep[3]=sqsum/size - (sum/size)$^2$; *non_zero = nonzero_count;
```

TABLE VI. Estimated Resource Usage Summary (Optimized Summary Statistics).

Resource	usage
Logic utilization	8%
Dedicated Logic registers	4%
Memory blocks	10%
DSP blocks	1%

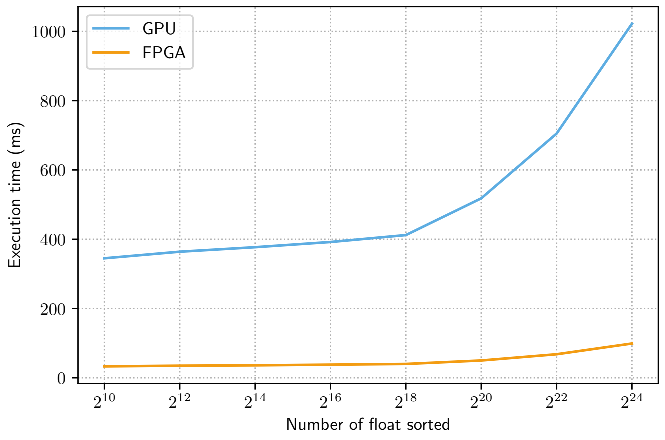

Fig. 5. Execution time of the Sort Algorithm for Many Input Sizes.

comparisons, it seems impossible to find a fixed organization of blocks that applies the algorithm to any input data. Whereas sorting networks, like Bitonic Sort present a complexity of $O\left(n\log^2(n)\right)$, seems highly optimized for FPGA, because they use a fixed data path. For the Bitonic sort algorithms, we configure the logic blocks to have two input and two output: the first one returning the max of the two input and the second one returning the minimum of these two. With these sorting networks, each new list to sort is computed every p clock cycles, p being the number of clock-cycle taken by logic block to return the max and min of the two inputs. In order to achieve a parallel implementation of Bitonic sort we divide the dataset onto multiple threads (one thread occupies at least one data). For each step of bitonic sort as we can notice in Fig. 2, all of comparing operations are executed simultaneously on available threads. Fig. 5 shows the performance evaluation of sort implementation on the GPU and FPGA. We run the GPU and FPGA implementation to sort a set of different float ranging from 2^{10} to 2^{24}. The obtained results shows the parallel computing power of the FPGA. For even large number of sorted float (2^{24}) the FPGA implementation is always efficient compared to the GPU implementation which need more then 1 second to process (2^{24}) floats.

C. OpenCL Implementations of K-means Algorithm

K-means has the difference of data independence from Bitonic Sort. In fact, to find the nearest centroid from one point, we can assign each points from data set onto available threads and perform the calculation and comparison; on the other hand, new centroids calculation needs data from the membership matrix, this step can be parallelized by assigning

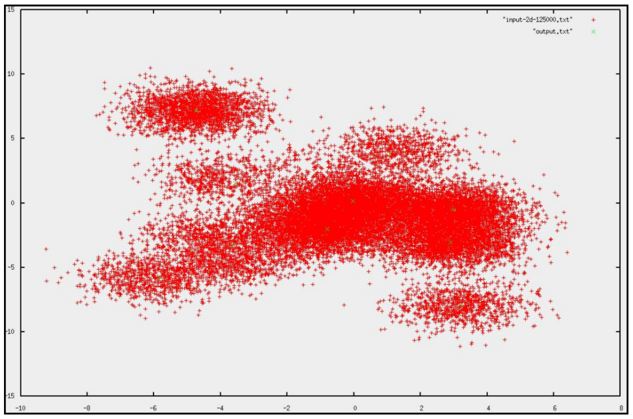

Fig. 6. K-mean GPU Result of Clustering 125000 2D-points into 10 Clusters.

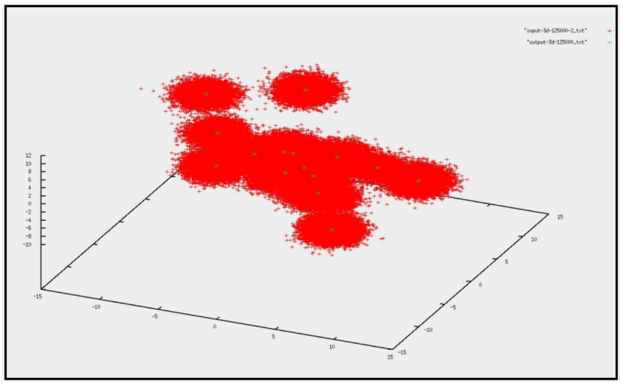

Fig. 7. K-mean FPGA Result of Clustering 125000 3D-points into 15 Clusters .

each cluster to a thread and carrying out the calculation separately for each of them. However in this implementation, we let the host device take the task sequentially. For K-mean implementation, the test data is generated by the Spark benchmark. Firstly, we tested the algorithm on small dimensional data (2D and 3D) in order to test the output. Then the algorithm is tested on 10 and 100 dimension points sets. Finally, we tested on high dimensional data (500D). The size of data ranged from 12500 to 125000 points for small dimensional sets and up to 31250 points on high dimensional sets (up to 300MB). Figures Fig. 6 and Fig. 7 shows respectively the GPU and FPGA results of the k-mean implementation. Each red point is the visualization of a point in the given data set and each green point is a centroid found by the heuristics. The obtained results confirm the functional validation about clustering. Indeed, the cost function (sum of square of the distance from each point to the centroid of the cluster containing it) is shown to decrease after each iteration.

Table VII shows the performance evaluation of k-mean implementation on the GPU and FPGA. The running time to cluster 125000 3D points is 380 ms for the GPU implementation and 33 ms for the FPGA implementation. For 31250 500D points is 12 seconds on the GPU and 158 ms on the FPGA. The occupancy achieved is 98.8 percent in the first measurement and 49.5 percent (over the theoretical 50 percent) in the second. This shows that the occupancy is well controlled. The GPU

TABLE VII. GPU AND FPGA K-MEAN EXECUTION TIME

Execution Times		
Platform	125 000 (3D)	31250 (500D)
GPU	380 (ms)	12 (s)
FPGA	33 (ms)	158 (ms)

implementation is far from real time performances. Indeed, the choice of block and grid size strongly affects the efficiency. For example, if the block size is reduced from (256 x 1 x 1) to (64 x 1 x 1), the time to find the centroids for the 125000 3D point set goes down from 380 ms to 250 ms. This can be explained by the use of synchronization in each block. By contrast, if the block size is increased from (61 x 1 x 1) to (1024 x 1 x 1) the running time on the data set of 31250 500D points goes down significantly from 12 seconds to 1.9 seconds. This can be explained by the large size of shared memory in each block. In the GPU implementation, the low DRAM utilization may come from the non-coalesced access to memory. Reducing this can also result in the better performance. The FPGA implementation is shown to outperform the GPU one. Only 158 ms is needed to cluster a very high dimensional data (500D).

D. OpenCL Implementations of Correlation Algorithms

The Pearson Correlation Algorithm was implemented using two reductions to find the mean of both input vectors, followed by the computation of the covariance of both entries and each one of the standard deviations. The final result is given by $cov\,(X;Y)\,/\,(\sigma_X * \sigma_Y)$. To calculate the Spearman coefficient we need to calculate the Pearson coefficient of the ranks. For the computation of the ranks, we first sort both samples by the values of the first one and calculate the ranks for the first sample by using a simple kernel that for each position position in the sample that has an element different from the next one, goes back and counts all the occurrences of that element and then finally fills all the position with that same value with the mean of the ranks. This part of the code is not very parallelizable and can run in $O\,(n)$ if all the elements in the initial sample are the same, opposed to $O\,(1)$ with all elements different from one another [23] . But since in normal samples with float values this is unlikely to happen the approach works well. Then we sort again the first, the second and the ranks of the first by the values of the second sample, calculate now the ranks of the second sample using the same method and now that we got both ranks array, we use the Pearson Correlation Coefficient algorithm in this data to find the Spearman Correlation Coefficient. For measuring the execution time of both correlation algorithms, we first generated 5 input files for different sizes of samples, then we ran each one of those input files and took the mean of the execution times for each one of the 5 input files. Fig. 8, Fig. 9 shows the performance evaluation respectively of the Pearson and Spearman Correlation algorithm implementation on the GPU and FPGA. We test both implementation using a number of elements in sample ranging from 2^{10} to 2^{24}. For a high number of elements 2^{24} the GPU process the pearson correlation algorithm in 1 seconds and the Spearman algorithm in 5.5 seconds. For the same number of elements, the FPGA implementation process the pearson and spearman correlation in 90 ms and 400 ms respectively. The processing time of the correlation algorithms on the FPGA implementation has been decreased by a factor x12 compared to the GPU implementation.

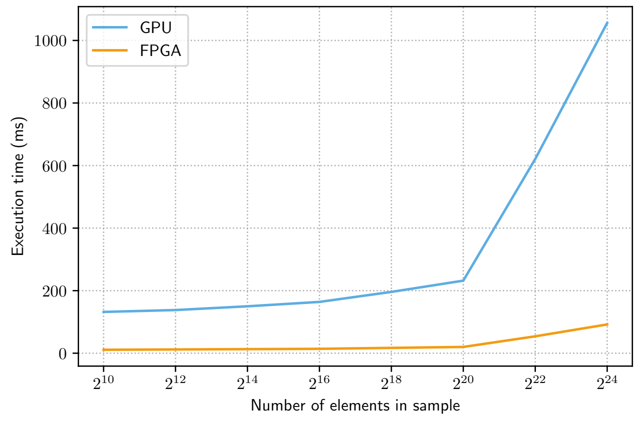

Fig. 8. Execution Time of the Pearson Algorithm for Many Input Sizes .

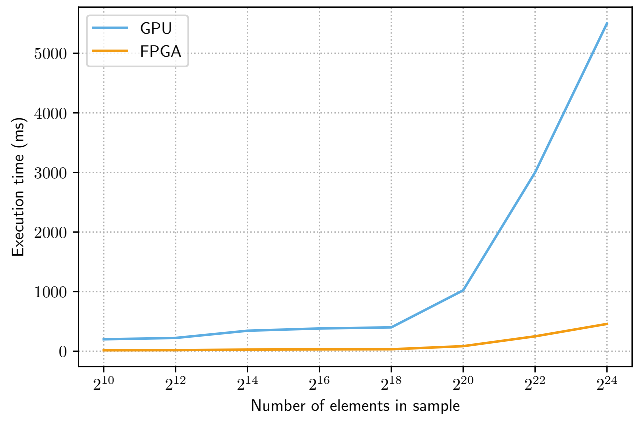

Fig. 9. Execution Time of the Spearman Algorithm for Many Input Sizes .

E. Synthetic Results

Our work adopted the high level synthesis for FPGA implementation using OpenCL. Despite of the advantage of high level programming, its use is still limited. The Intel offline compiler takes a lot of time in order to generate the hardware configuration files (aocx) (duration of compilation hours for more complex function and if more optimization are requested from the compiler). However, we can achieve higher acceleration using OpenCL, which provides better memory management. We can freely access the local, global and constant memory in the OpenCL kernel. This allows us to better manage the data transmission and data structure. In addition, the FPGA is considered as the generation of integrated circuit claimed higher performance and reliability, and the emerging high level software tools make it easily accessible to the community. The encouraging results we obtained on the FPGA in term of acceleration performance, demonstrates that a dedicated architecture can be used to prototype a micro-server for big data that operates under real-time constraints. As a future work, we intend to achieve a full embedded implementation for Big Data algorithms on FPGA using the integrated ARM processor of the Arria 10 SoC.

VII. Conclusion

In this work, we have implemented and optimized three algorithms: Bitonic Sorting network, K-means, Spearman and Pearson correlation. The purpose behind this implementation is to prototype a micro-server for processing big data algorithms on both GPU and FPGA and compare their performance. We have implemented and quantitatively evaluated the execution times of some of the most important algorithms for big data. We present performance results on a heterogeneous architectures: high-end CPU-GPU and a dedicated CPU-FPGA architecture. The choice of using dedicated architecture was made principally because the big data algorithms can be massively parallelized. This property is exploited by using a dedicated FPGA-based architecture as a target platform for an efficient embedded micro-server. The performance of the optimized algorithms on the FPGA show a promising prospect of utilizing them in solving real-world problems.

References

[1] A. K. Tiwari, H. Chaudhary, and S. Yadav, "A review on big data and its security," in *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*. IEEE, 2015, pp. 1–5.

[2] D. Laney *et al.*, "3d data management: Controlling data volume, velocity and variety," *META group research note*, vol. 6, no. 70, p. 1, 2001.

[3] K. Neshatpour, M. Malik, M. A. Ghodrat, and H. Homayoun, "Accelerating big data analytics using fpgas," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 164–164.

[4] C.-C. Chung, C.-K. Liu, and D.-H. Lee, "Fpga-based accelerator platform for big data matrix processing," in *2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*. IEEE, 2015, pp. 221–224.

[5] C. Wang, X. Li, and X. Zhou, "Soda: Software defined fpga based accelerators for big data," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 884–887.

[6] J. Hoozemans, J. Peltenburg, F. Nonnemacher, A. Hadnagy, Z. Al-Ars, and H. P. Hofstee, "Fpga acceleration for big data analytics: Challenges and opportunities," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 30–47, 2021.

[7] J. Hou, Y. Zhu, L. Kong, Z. Wang, S. Du, S. Song, and T. Huang, "A case study of accelerating apache spark with fpga," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2018, pp. 855–860.

[8] O. Debauche, S. A. Mahmoudi, S. Mahmoudi, and P. Manneback, "Cloud platform using big data and hpc technologies for distributed and parallels treatments," *Procedia Computer Science*, vol. 141, pp. 112–118, 2018.

[9] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 13–24.

[10] Y. Censor, "Pareto optimality in multiobjective problems," *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977.

[11] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[12] A. Ahmad and L. Dey, "A k-mean clustering algorithm for mixed numeric and categorical data," *Data & Knowledge Engineering*, vol. 63, no. 2, pp. 503–527, 2007.

[13] R. Ostrovsky, Y. Rabani, L. J. Schulman, and C. Swamy, "The effectiveness of lloyd-type methods for the k-means problem," *Journal of the ACM (JACM)*, vol. 59, no. 6, pp. 1–22, 2013.

[14] P. S. Bradley and U. M. Fayyad, "Refining initial points for k-means clustering." in *ICML*, vol. 98. Citeseer, 1998, pp. 91–99.

[15] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 307–314.

[16] K. J. Liszka and K. E. Batcher, "A generalized bitonic sorting network," in *1993 International Conference on Parallel Processing-ICPP'93*, vol. 1. IEEE, 1993, pp. 105–108.

[17] M. T. Goodrich, "Randomized shellsort: A simple oblivious sorting algorithm," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2010, pp. 1262–1277.

[18] I. Parberry, "The pairwise sorting network," *Parallel Processing Letters*, vol. 2, no. 02n03, pp. 205–211, 1992.

[19] D. E. Knuth, "Sorting and searching," 1973.

[20] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[21] L. Myers and M. J. Sirois, "Spearman correlation coefficients, differences between," *Encyclopedia of statistical sciences*, vol. 12, 2004.

[22] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[23] S. Kim, M. Ouyang, and X. Zhang, "Compute spearman correlation coefficient with matlab/cuda," in *2012 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. IEEE, 2012, pp. 000 055–000 060.